VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vylepšení metaheuristických algoritmů pomocí symbolické regrese

Improvement of Metaheuristic Algorithms Using Symbolic Regression

2020

Bc. et Bc. Vladimíra Černá

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. et Bc. Vladimíra Černá**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Vylepšení metaheuristických algoritmů pomocí symbolické regrese
Improvement of Metaheuristic Algorithms Using Symbolic Regression

Jazyk vypracování: čeština

Zásady pro vypracování:

Metaheuristické algoritmy nám pomáhají řešit problémy, které jsou běžnými metodami obtížně řešitelné. Jako zástupce takovýchto algoritmů můžeme uvést samoorganizující se migrační algoritmus, optimalizaci hejnem částic, či diferenciální evoluci. Vědci se stále snaží vylepšovat tyto algoritmy tak, aby tyto algoritmy dosahovaly lepších výsledků v kratším čase. Úkolem této práce je vybrat vhodné algoritmy a pokusit se je vylepšit pomocí symbolické regrese, např. syntézou nového šumového vektoru, syntézou zpětnovazebního řízení, nebo syntézou adaptivních parametrů.

Úkoly této práce se dají shrnout v následujících bodech:
1. Nastudovat problematiku metaheuristických algoritmů a symbolické regrese.
2. Naprogramovat alespoň 3 vhodné metaheuristické algoritmy.
3. Naprogramovat vhodný algoritmus pro symbolickou regresi.
4. Navrhnout možnosti vylepšení zvolených algoritmů pomocí symbolické regrese a tato vylepšení naprogramovat.
5. Porovnat původní algoritmy s novými.

Seznam doporučené odborné literatury:

[1] ZELINKA, Ivan, et al. Evoluční výpočetní techniky: principy a aplikace. BEN, 2008.
[2] DAVENDRA, Donald, et al. Self-organizing migrating algorithm. New optimization techniques in engineering, 2016.
[3] SHI, Yuhui, et al. Particle swarm optimization: developments, applications and resources. In: Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546). IEEE, 2001. p. 81-86.
[4] DAS, Swagatam; MULLICK, Sankha Subhra; SUGANTHAN, Ponnuthurai N. Recent advances in differential evolution–an updated survey. Swarm and Evolutionary Computation, 2016, 27: 1-30.
[5] ZELINKA, Ivan; OPLATKOVA, Zuzana; NOLLE, Lars. Analytic programming–Symbolic regression by means of arbitrary evolutionary algorithms. Int. J. of Simulation, Systems, Science and Technology, 2005, 6.9: 44-56.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **prof. Ing. Ivan Zelinka, Ph.D.**

Datum zadání: 01.09.2019

Datum odevzdání: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Ostravě 15.5.2020 .....

Ráda bych na tomto místě poděkovala vedoucímu této diplomové práce prof. Ing. Ivanu Zelinkovi, Ph.D. za poskytnutí cenných rad a podkladů, bez kterých by tato práce nevznikla. Ráda bych také poděkovala svému příteli Ing. Matouši Jandekovi, jenž mi byl během vzniku této práce velkou oporou.

Abstrakt

Jelikož na poli optimalizace stále dochází k vývoji a mnohým výzkumům, je cílem této práce nalezení zlepšení metaheuristických algoritmů SOMA, PSO a DE prostřednictvím analytického programování. Proto se tato práce v prvních částech zabývá rozborem těchto metaheuristických algoritmů a jsou zde také popsány principy analytického programování, jež bylo využito jako metoda symbolické regrese. Všechny algoritmy byly posléze implementovány v jazyce C++ pro účely experimentů, jejichž výsledky jsou poté prezentovány a vyhodnoceny v závěrečných částech. Zlepšení optimalizačních schopností se podařilo nalézt především u algoritmu SOMA. U algoritmů PSO a DE došlo ke zlepšení u vybraných testovacích funkcí.

Klíčová slova: metaheuristické algoritmy; optimalizace hejnem částic; samoorganizující se migrační algoritmus; diferenciální evoluce; symbolická regrese; analytické programování;

Abstract

Optimization methods are still under development, and researchers are working on the improvement of current methods. The purpose of this thesis is to find an improvement of three metaheuristic algorithms - SOMA, PSO, and DE. The analytic programming is used as a method of symbolic regression for this purpose. The beginning of this thesis consists of descriptions of SOMA, PSO, and DE, as well as of analytic programming. All algorithms were implemented in the C++ programming language and experiments were performed. The results are evaluated at the end of this thesis. Significant improvement was found for the SOMA algorithm. For PSO and DE, improvements were observed for some of the objective functions.

Keywords: Metaheuristic Algorithms; Particle Swarm Optimization; Self-Organizing Migrating Algorithm; Differential Evolution; Symbolic Regression; Analytic Programming

Obsah

Seznam použitých zkratek a symbolů

SOMA	–	Samoorganizující se migrační algoritmus, z anglického Self-Organizing Migrating Algorithm
PRT	–	Perturbace
D	–	Dimenze
MOSOMA	–	Samoorganizující se migrační algoritmus pro víceúčelovou optimalizaci, z anglického Multi-Objective Self-Organizing Migrating Algorithm
DSOMA	–	Diskrétní samoorganizující se migrační algoritmus, z anglického Discrete Self-organising Migrating Algorithm
PSO	–	Optimalizace hejnem částic, z anglického Particle Swarm Optimization
INPSO	–	Optimalizace hejnem částic s nezávislými sousedstvími, z anglického Independent Neighbourhood Particle Swarm Optimization
CSPSO	–	Spojení PSO s algoritmem kris-kros, z anglického Crisscross Search Particle Swarm Optimization
CSO	–	Spojení civilizačního algoritmu s PSO, z anglického Civilized Swarm Optimization
GPEA	–	Z anglického Geometrical Place Evolutionary Algorithms
DE	–	Diferenciální evoluce
SaDE	–	Samo-adaptivní diferenciální evoluce, z anglického Self Adaptive Differential Evolution
jDE	–	Druh samo-adaptivní diferenciální evoluce
JADE	–	Samo-adaptivní diferenciální evoluce s externím archivem
SHADE	–	Samo-adaptivní diferenciální evoluce založena na předchozím úspěchu
CBPI	–	Inicializace populace založena na shlukování, z anglického Clustering Based Population Initialization
DSH	–	Ošetření celočíselného parametru jedince, z anglického Discrete Set Handling
GFS	–	Množina neterminálů a terminálů pro sestavení výrazu, z anglického General Function Set
GCC	–	Kompilátor, z anglického GNU Compiler Collection
IDE	–	Vývojové prostředí, z anglického Integrated Development Environment
SW	–	Program, z anglického Software
AP	–	Analytické programování

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Celá naše planeta funguje na principu optimalizace, neboť jen to, co je optimální v daných podmínkách, dokáže přežít. Ať již hovoříme o přírodních útvarech či živých organismech. A jelikož je člověk nedílnou součástí koloběhu života této planety, také on se odpradávna snaží nalézt nejoptimálnější řešení různých situací, se kterými se během svého bytí potýká. Činíme tak často nevědomky i zcela uvědoměle. Proto se toto téma stalo námětem mnohých výzkumů a testů v mnoha oblastech [1, 2, 3, 4, 5, 6]. Také v informatice tomu není jinak. A pokud se spojí informatika s ději v přírodě, pak vznikají algoritmy, které z každé oblasti čerpají její výhody [7]. Touto inspirací vznikla celá řada algoritmů pro stochastickou optimalizaci systémů, kde není předem jasný vztah mezi vstupními parametry a výstupními hodnotami, a také všude tam, kde by klasické metody poskytovaly časově náročné řešení nebo za předpokladu, kdy by jejich použití selhávalo úplně [1, 7]. Každý si jistě v životě položil otázku, ať už se tak stalo během každodenního rutinního úkonu či zcela nové činnosti, zda způsob, jakým věci vykonává, je již dostatečně optimální, nebo by se dal dále vylepšovat. Jelikož se ve svém životě snažíme věci tzv. posouvat kupředu, je toto téma diplomové práce velmi aktuální, noří se do tajů optimalizačních metod inspirovaných přírodou a zabývá se následnou otázkou, zda je možné vybrané zástupce dále optimalizovat pro daný problém. Celá práce se dělí na tři části - teoretický úvod, popis implementace a shrnutí nalezených výsledků.

V teoretické části jsou obsaženy kapitoly zabývající se teoretickým rozborem později implementovaných algoritmů. První sekce této práce je tedy věnována algoritmu SOMA, principu jeho fungování, parametrům, jež ovlivňují jeho výkonnost, a také variantám, které v průběhu času vznikaly. Další sekce rozebírá algoritmus PSO, jeho typické fáze, jimiž algoritmus prochází, parametry, jež ovlivňují jeho optimalizační schopnosti, a jeho varianty. Posledním optimalizačním algoritmem, jež je zde popsán, je DE. Tato část má stejnou strukturu jako předešlé dvě. Poslední sekce teoretické části této práce je věnována symbolické regresi a jedné z jejich metod - analytickému programování. Na příkladech jsou také vysvětleny jeho principy a nejčastější problémy, které mohou v jeho průběhu nastat.

Druhá část, jež je věnována praktické implementaci řešení, popisuje ve svých úvodních částech nástroje, jež byly k implementaci využity, a jejich verze. Tato část je následována popisem implementace testovacích funkcí využitých pro testování optimalizačních schopností jednotlivých metaheuristických algoritmů. Následující části jsou pak věnovány implementaci tří zmíněných metaheuristických algoritmů - SOMA, PSO a DE. Další část se věnuje praktické implementaci analytického programování. V navazujících kapitolách jsou poté popsány nastavení, s nimiž lze program spustit a také vlastní pomocné knihovny, jež byly implementovány a jsou v různých částech kódu využívány.

Závěrečná část této práce je věnována experimentům, jež byly provedeny s cílem vylepšení metaheuristických algoritmů. Jsou popsány také hodnoty parametrů, jež byly využity pro jednotlivé algoritmy. Následuje pak shrnutí nalezených výsledků a jejich statistické vyhodnocení.

Význam nalezených řešení je pak shrnut a zdůvodněn v závěru této práce.

2 Samoorganizující se migrační algoritmus

V této části textu bude představen algoritmus s názvem Samoorganizující se migrační algoritmus (SOMA) [1, 8, 9], jenž se inspiroval chováním skupiny jedinců v sociálně-biologickém prostředí. Během jedné iterace zvané migrační kolo dochází k postupné změně polohy jedinců. Jedinci jsou na začátku náhodně generováni podle prototypu, jež udává datový typ parametrů jedince a jejich rozsah. SOMA není přímo založena na evolučních teoriích, tzn. že z rodičů vznikne potomek, jež tvoří další generací, ale spíše na sociálním chování jedinců uvnitř skupiny. Toto charakterizuje algoritmy hejnové či memetické. Ačkoliv v průběhu každé iterace u SOMA nedochází k vytváření nové generace jedinců, nýbrž k jejich pohybu, stále probíhá selekce výběru nejlepšího jedince, a tedy je možné tento algoritmus řadit také mezi evoluční algoritmy. Zde je také řazena svým autorem [1]. Příkladů chování živých organismů, které algoritmus SOMA inspirovaly, lze v přírodě nalézt nemálo. Jedinci vždy stanoví nejúspěšnějšího z nich a postupným prohledáváním se v jednotlivých krocích k němu blíží. V následujících kapitolách budou jednotlivé fáze algoritmu rozebrány detailněji.

2.1 Obecné principy SOMA

Pro odlehčení textu uvedu příklad z reálného světa, pro který nemusím chodit daleko, neboť jsem hrdou majitelkou dvou loveckých psů a přesně to chování, jenž každý text zabývající se optimalizační technikou SOMA popisuje hned v úvodu, je velmi dobře pozorovatelné také na chování mé psí smečky. Pro jejich vyžití a mé odreagování často provozujeme lovecké techniky jako je například slídění nebo dohledávka pernaté či srstnaté zvěře v terénu. Pokud vypustím oba psy najednou, pak následuje vždy stejný průběh - oba psi charakteristickým „cikcak“ pohybem prohledávají krajinu sledujíc jeden druhého. V momentě, kdy jeden z psů zaznamená podnět, který by mohl znamenat úspěšné dohledání, druhý pes, opět „cikcak“ pohybem, běží k prvnímu, avšak stále dohledává. Pokud při své cestě najde zajímavější pach, nežli pes první, první pes okamžitě mění směr k druhému úspěšnějšímu psovi. Je třeba podotknout, že se mi dosud za celých 6 let nestalo, že bych pro odložený kus musela dojít sama, ačkoliv psi nikdy předem neví, kde se zvěř nachází. A právě toto chování algoritmus SOMA napodobuje a jeho náznaky jsou v něm velmi patrné, sami posuďte v následující části.

Jak již bylo řečeno, SOMA se inspirovala v přírodě, a proto také využívá některé analogie. Tyto jsou uvedeny v Tabulce 1.

SOMA pracuje jako mnoho optimalizačních algoritmů v cyklech, kterým se v tomto případě říká migrační kola. Tato jsou analogií ke generacím v evolučních algoritmech. Termín generace je užíván z toho důvodu, že v každém kole evolučních algoritmů vznikne sada nových jedinců s využitím genů dvou a více jedinců ze sady předchozí. Toto u SOMA neprobíhá. V každém kole dochází ke změně polohy postupně všech jedinců směrem k nejlepšímu z nich tzv. lídrovi.

Základní běh algoritmu SOMA probíhá v následujících krocích:

Tabulka 1: Analogie převzaté z reálného světa v SOMA

Reálný svět	Analogie v SOMA
členové hejna či smečky	jedinci v populaci
jedinec, jenž našel nejlepší zdroj	lídr neboli nejlepší jedinec v populaci
kvalita zdroje	ohodnocení v N-dimenzionálním prostoru
životní prostor smečky	N-dimenzionální prostor
pohyb smečky v životním prostoru	pohyb jedinců v SOMA algoritmu

1) Definování parametrů. Před započetím algoritmu je třeba, aby byly určeny všechny parametry, jež jsou uvedeny v Tabulce 2.

2) Vytvoření počáteční populace. Jednotliví jedinci populace jsou tvořeni náhodně tak, aby odpovídali typovému jedinci a leželi uvnitř oblasti, která je vyšetřována, tzn. jejich souřadnice byly v mezích [*min, max*]. Na konci tohoto kroku jsou všichni jedinci ohodnoceni s využitím účelové funkce.

3) Migrace. V tomto kroku je určen lídr celé populace - jedinec s nejlepším ohodnocením. Následuje fáze, kdy jeden jedinec po druhém skáče k lídrovi po krocích. Po každém kroku je provedeno ohodnocení nové pozice jedince. Jakmile jedinec svou cestu tvořenou jednotlivými skoky k lídrovi dokončí, je přesunut do pozice, kde měl nejlepší ohodnocení.

SOMA při výpočtu nové polohy jedince po každém skoku využívá operátor, který má stejný význam jako mutace v evolučních algoritmech, v tomto případě se však nazývá perturbační vektor *(PRTvektor)* od anglického slova perturbation - odchylka. Tento vektor zajišťuje potřebnou diverzitu v pohybu jedince v průběhu jedné iterace. Před každým skokem jedince je generováno náhodné číslo, které je porovnáno s parametrem *PRT*. Pokud je toto číslo menší než hodnota *PRT*, pak je odpovídající hodnota perturbačního vektoru nastavena na hodnotu 1, v opačném případě pak na hodnotu 0. Tímto postupem dochází k odchylkám od přímočarého pohybu jedince k lídrovi ukotvením jednotlivých souřadnic jedince. Toto si lze vizualizovat jako jakési uskakování z cesty, případně si lze představit pohyb psa při slídění, jak bylo zmíněno v předchozí části. Ke generování perturbačního vektoru dochází před každým dílčím skokem jedince, proto se odchylky v jednotlivých dimenzích v každém skoku mohou lišit. Experimentálně bylo objeveno, že SOMA bez využití *PRTvektoru* inklinovala spíše k nacházení lokálního extrému namísto globálního.

Migrace lze považovat za soutěživě-kooperativní chování, neboť v průběhu jednoho migračního kola se jedinci snaží najít nejlepší pozici z hlediska ohodnocení vůči ostatním i vůči svým doposud navštíveným pozicím v průběhu migrace. Toto by se dalo vnímat jako fáze soutěžení. Fáze kooperace by pak následovala v okamžiku, kdy dojde ke sdělení pozic jednotlivých jedinců všem ostatním na konci migrace a je vybrán nový lídr. Soutěživě-kooperativní chování je jedním ze znaků memetických či hejnových algoritmů, a právě z tohoto důvodu zde bývá SOMA taktéž řazena.

4) Kontrola ukončovacích podmínek. V tomto kroku proběhne kontrola, zda rozdíl mezi lídrem a nejhorším jedincem není menší než parametr zadaný na začátku popřípadě, zda nedošlo k provedení počtu migrací, jež byl opět zadán jako vstupní parametr. Pokud nejsou splněny podmínky pro ukončení, následuje návrat ke kroku 3, v opačném případě algoritmus přejde k následujícímu kroku 5.

5) Ukončení algoritmu. Zde se již provede vrácení nejlepšího nalezeného řešení.

Princip algoritmu SOMA *AllToOne* by se dal shrnout následujícím pseudokódem.

Vstupy:
P^0 - počáteční generace
$PathLength$ - řídicí parametr
$Step$ - řídicí parametr
PRT - řídicí parametr
$PopSize$ - řídicí parametr
$MinDiv$ - ukončovací parametr
$Migrations$ - ukončovací parametr
$SpeciMan$ - prototypový jedinec
f_{cost} - účelová funkce
generuj náhodnou počáteční populaci P^0
for $i < Migrations$ **do**
  vyber nejlepšího jedince - lídra
  for $j < PopSize$ **do**
    vyber P_j
    realizuj pohyb P_j a vždy ohodnoť pomocí f_{cost}
    nejlepší polohu jedince vyber do dalšího migračního kola
  end for
  if (P_{best} - P_{worst}) $< MinDiv$ **then**
    vrať P_{best}
    ukonči algoritmus
  end if
end for
vrať P_{best}

kde v tomto algoritmu i představuje jednotlivá migrační kola a platí $i = 0...Migrations$, j představuje pořadí jedince v populaci a platí $j = 1...PopSize$, P_j představuje jedince v populaci P na $j-té$ pozici, P_{best} je pak nejlepší jedinec v populaci P a P_{worst} je pak nejhorší jedinec v populaci P. V této práci nebyl využit parametr $MinDiv$. Jako ukončovací parametr byl volen pouze parametr $Migrations$, tedy počet migračních kol algoritmu.

2.2 Fáze SOMA algoritmu

V následující části budou popsány jednotlivé fáze a operace, které jsou v SOMA postupně prováděny, a budou blíže charakterizovány.

2.2.1 Populace

SOMA je jako ostatní hejnové a evoluční algoritmy založena na populaci. Populace je složená z jedinců, přičemž jedince si můžeme představit jako konfiguraci jednotlivých parametrů, jež tvoří vstup pro účelovou funkci. Vhodnost jedince je pak určena právě pomocí účelové funkce. Na počátku je náhodně generovaná populace. Aby k tomuto mohlo dojít, je potřeba znát ještě před začátkem algoritmu typového jedince. Ten udává, jakého datového typu budou parametry jedinců a jejich minimální a maximální hodnotu. Typový jedinec je popsán v (1).

$$typový\ jedinec = \{\{\mathbb{R}, [min, max]\}, \{\mathbb{Z}, [min, max]\}, ...\}. \tag{1}$$

Určení vhodných hranic pro jednotlivé parametry je velmi důležitý krok, neboť pokud budou hranice zvoleny nevhodně, výsledek optimalizace může být neaplikovatelný v reálném světě. Například v případě, kdy bychom dostali záporný průřez nosníku nebo zápornou tloušťku zdi. Jelikož mé studijní výsledky pocházejí také z fakulty stavební, dovolím si uvést dnes velmi aktuální příklad minimalizace průměrného součinitele prostupu tepla U_{em}, jež je závislý na součiniteli prostupu tepla konstrukcí U_{kon}, součiniteli prostupu tepla oken U_w, poměru plochy oken ku ploše fasády F_{gla} a poměru plochy fasády vůči celkové ochlazované ploše stavby F_{fas}. Každý ze jmenovaných parametrů může nabývat pouze určitých hodnot (hodnoty vychází z návrhu na základě požadavků klienta a také norem). Například dům bez oken nebo konstrukční materiál se součinitelem prostupu tepla blízkým nule jsou nereálné případy [10].

Po určení vhodných hranic pro jednotlivé parametry následuje tvoření jedinců pomocí náhodně generovaného čísla dle (2).

$$x^0_{i,j} = x_{min,j} + rand_{i,j} \cdot (x_{max,j} - x_{min,j}), \tag{2}$$

kde $x^0_{i,j}$ značí, že se jedná o počáteční populaci, index i má význam pořadí generovaného jedince a index j symbolizuje číslo parametru, který je generován, $x_{max,j}$ značí jedince, jež leží na horní hranici prohledávaného prostoru možných řešení, a analogicky pak $x_{min,j}$ představuje jedince, jež leží na dolní hranici prohledávaného prostoru možných řešení.

2.2.2 Perturbace

Jedná se o náhodné odchylky jedinců, které pomáhají udržovat diverzitu populace a také do populace zanáší zpět informaci, která byla v jednotlivých fázích algoritmu ztracena. U algoritmu SOMA je perturbace zavedena prostřednictvím PRT parametru, který do pohybu jedince vnáší odchylky. Parametr PRT může nabývat hodnot [0,1] a je použit pro vytvoření $PRTVektoru$.

Ten je pak tvořen dle (3).

$$PRTVector_i = \begin{cases} 1 & \text{pokud } rand_i < PRT, \\ 0 & \text{jinak.} \end{cases} \tag{3}$$

Tento přístup je inovativní v tom, že $PRTVektor$ je generován ještě před započetím pohybu jedince k lídrovi. Díky $PRTVektoru$ není pohyb jedince přímočarý, je tomu tak z důvodu, že dojde k zmrazení určitých souřadnic jedince. Později v průběhu vývoje tohoto algoritmu došlo k tomu, že je $PRTVektor$ generován před každým skokem, což částečně vylepšilo výkonnost algoritmu. V takovém případě trajektorie pohybu nabývá složitějších tvarů.

2.2.3 Křížení

Ve standardních evolučních algoritmech křížení probíhá tak, že z několika jedinců populace je vytvořen nový jedinec populace následující. Pokud bychom se dívali na tento krok z pohledu geometrie, pak jsou vybrány nové souřadnice v prohledávaném prostoru. V algoritmu SOMA, který je založen na kooperativním jednání inteligentních jedinců, je vybrána posloupnost nových pozic v N-dimenzionálním prohledávaném prostoru. V případě klasických evolučních algoritmů by tyto pozice představovaly jedince vzniklé operací křížení. Výpočet pozic jedinců je dán následující rovnicí:

$$x_i^{ML+1} = x_{i,start}^{ML} + (x_L^{ML} - x_{i,start}^{ML}) \cdot t \cdot PRTVector_i, \tag{4}$$

kde i označuje pořadí jedince v populaci, ML pořadí migračního kola, x_i^{ML+1} novou polohu jedince, $x_{i,start}^{ML}$ startovní polohu jedince, x_L^{ML} polohu lídra, $PRTVector_i^{ML}$ značí perturbační vektor pro daného jedince a dané migrační kolo a pro t platí, že $t \in [0, \text{po } Step \text{ az po}, PathLength]$.

2.3 SOMA parametry

Tak jako ostatní hejnové a evoluční algoritmy také SOMA ke svému běhu využívá několik parametrů. Některé z nich jsou zodpovědné za ukončení algoritmu v případě, že dojde k naplnění jedné nebo více podmínek. Nazýváme je tedy parametry ukončovací. Další skupinu parametrů tvoří ty, jež ovlivňují kvalitu výsledku, který je vrácen na konci běhu algoritmu, neboli parametry dané účelovou funkcí. Pro lepší přehlednost jsou parametry umístěny do Tabulky 2.

2.3.1 Parametr *PathLength*

Tento parametr udává, v jaké vzdálenosti za lídrem se jedinec zastaví. Pokud bychom například volili hodnotu 1, jedinec se zastaví přesně na pozici lídra. Pokud by tento parametr nabýval hodnoty 2, pak by se jedinec zastavil ve stejné vzdálenosti za lídrem, jako původně začínal na opačné straně. Pokud bychom tento parametr volili jako čísla menší než 1, pak by se jedinec zastavil ještě před lídrem, v takovém případě má SOMA tendence k předčasné konvergenci a uvíznutí v lokálním extrému.

Tabulka 2: Parametry SOMA podle [1]

Parametr	Druh	Doporučené hodnoty
$PathLength$	řídicí	[1,1; 5,0]
$Step$	řídicí	[1,1; $PathLength$]
PRT	řídicí	[0; 1]
D	počet parametrů v objektové funkci	dáno problémem
$PopSize$	řídicí	>10
$Migrations$	ukončovací	>10
$MinDiv$	ukončovací	určuje uživatel

2.3.2 Parametr *Step*

Tento parametr udává, s jakou granularitou má být prohledávána oblast, kterou jedinec cestuje směrem k lídrovi. V případě, že je účelová funkce unimodální jednoduchá funkce, je možné za tento parametr volit vyšší hodnoty, abychom urychlili konvergenci algoritmu. Pokud však není předem znám průběh účelové funkce, je vhodnější volit doporučené hodnoty. Není doporučeno, aby hodnota tohoto parametru byla celočíselným násobkem parametru $PathLength$. Proto je doporučená hodnota 0,11.

2.3.3 Parametr *PRT*

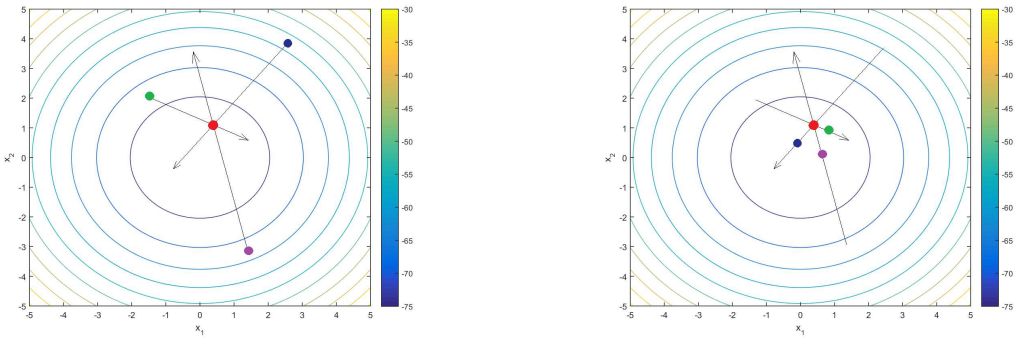
Tento parametr udává, jakých odchylek se jedinec bude dopouštět v průběhu své cesty k lídrovi. Je to jeden z nejcitlivějších parametrů SOMA algoritmu. Doporučuje se volit hodnoty blízké 0,1. Čím vyšších hodnot tento parametr nabývá, tím rychleji algoritmus konverguje. Pokud je účelová funkce málodimezionální a počet jedinců volíme vysoký, pak lze použít hodnoty 0,7 až 1,0. V případě krajní hodnoty 1,0 se vytrácí stochastická vlastnost algoritmu a přechází v algoritmus deterministický podobný lokálnímu prohledávání.

2.3.4 Parametr *D*

Dimenze je dána problémem, který má být optimalizován. Jedná se o počet optimalizovaných proměnných, jejichž vzájemný vztah udává účelová funkce. Tato hodnota nemůže být uživatelem měněna za podmínky, že nedojde k reformulaci problému a tedy ke změně počtu parametrů.

2.3.5 Parametr *PopSize*

Jde o parametr udávající velikost populace. Doporučení pro tuto hodnotu je, aby nabývala hodnot přibližně mezi 0,2 a 0,5 dimenze problému (parametru D). V případě, že je účelová funkce unimodální, pak postačí nižší počet jedinců než pro multimodální variantu. V takovém případě je vhodné volit počet jedinců blízký dimenzi problému. Obecně však platí, že pokud je



(a) Stav před migrací

(b) Stav po migraci

Obrázek 1: Algoritmus SOMA strategie AllToOne

počet jedinců menší než 10, pak výkonnost SOMA klesá, proto je tedy doporučeno volit velikost populace vyšší než 10.

2.3.6 Parametr *Migrations*

Tento parametr udává, kolik migrací jedinců má proběhnout. Jedná se o analogii k počtu generací u DE či GA. V případě algoritmu SOMA se tento běhový cyklus nazývá migrace, aby lépe vystihoval podstatu algoritmu, tedy pohyby jednotlivých jedinců v prohledávaném prostoru. Je doporučeno volit hodnoty vyšší než 10.

2.3.7 Parametr *MinDiv*

Jedná se o ukončovací kritérium algoritmu. Udává míru rozdílu mezi lídrem a jedincem. Pokud je tento rozdíl menší než $MinDiv$, pak je algoritmus ukončen. V případě, že je tento parametr volen jako záporné číslo, pak nemá na běh algoritmu žádný vliv.

2.4 SOMA strategie

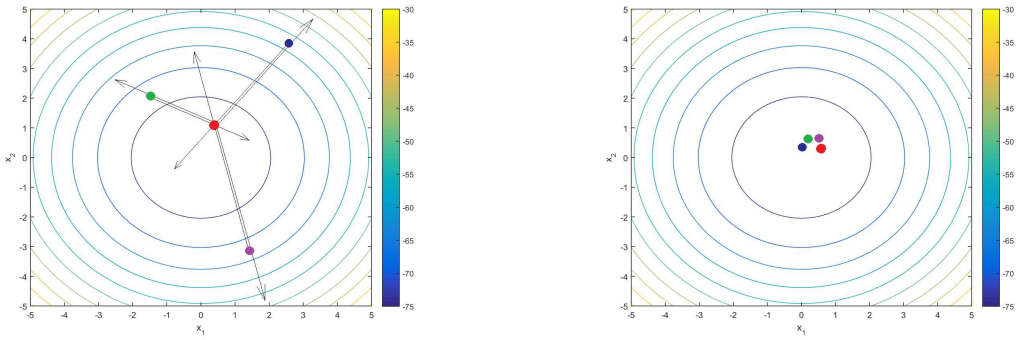
Tak jako většina optimalizačních algoritmů také SOMA má své varianty a strategie výpočtu. Ty budou popsány v následujících částech této práce.

2.4.1 Strategie *AllToOne*

Jedná se o základní postup, který byl popsán v předchozích kapitolách. Všichni jedinci se postupně pohybují k lídrovi, který setrvává na své pozici. Tento pohyb znázorňuje Obrázek 1.

2.4.2 Strategie *AllToAll*

V této strategii putuje každý jedinec postupně ke všem ostatním jedincům. Před započetím pohybu k dalšímu jedinci vždy vychází ze své původní pozice, avšak historii pozic si uchovává,



(a) Stav před migrací

(b) Stav po migraci

Obrázek 2: Algoritmus SOMA strategie AllToAll

aby z nich pak mohla být vybrána ta nejlepší na základě ohodnocení účelovou funkcí. Jedná se o modifikaci, která je náročnější na výpočetní výkon, nicméně k tomu, aby se algoritmus dostal ke správnému optimu, je potřeba menšího počtu migrací oproti *AllToOne* strategii, neboť každý jedinec vystřídá větší množství pozic. Postup při *AllToAll* strategii je zachycen na Obrázku 2.

2.4.3 Strategie *AllToAll Adaptive*

Tato strategie je velmi podobná strategii *AllToAll*, avšak s rozdílem, že jedinec nezačíná každé migrační kolo ze své původní pozice, jak je tomu v *AllToAll*, avšak z nejlepší pozice, kterou našel v průběhu putování k předchozímu jedinci.

2.4.4 Strategie *AllToRand*

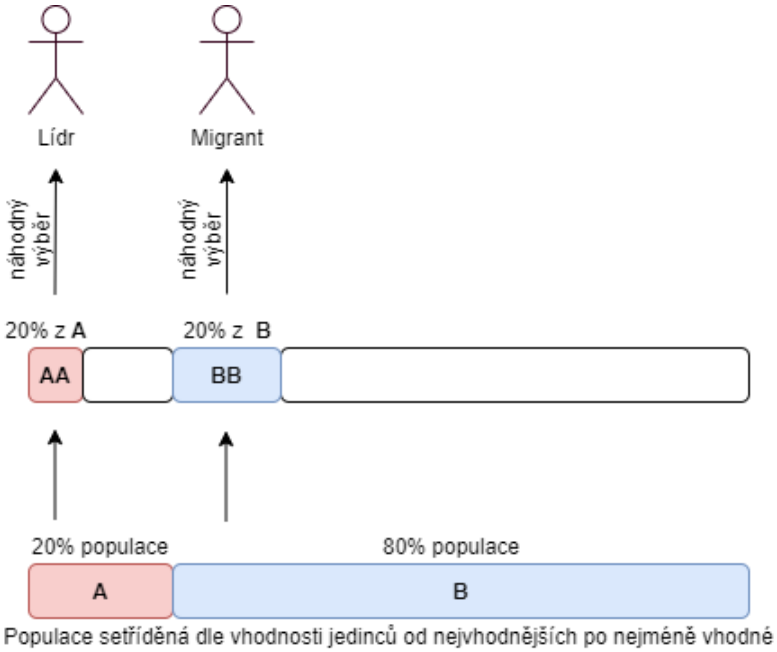
V této strategii v jednom migračním kole jedinec cestuje k náhodně zvolenému jedinci bez ohledu na jeho vhodnost. Náhodně vybraných jedinců může být více, jejich počet záleží na volbě uživatele. V zásadě se zde nabízí dvě další dílčí strategie - počet náhodně vybraných jedinců je předem dán a nemění se, nebo je každé migrační kolo vybrán náhodně jiný počet jedinců, ke kterým bude jedinec postupně putovat.

2.5 Další varianty SOMA

V následující části budou jen krátce představeny novější varianty SOMA algoritmu.

Za zmínku jistě stojí také SOMA s využitím Paretova principu [3, 11]. Zde v první fázi je stejně jako v ostatních verzích generovaná náhodná počáteční populace jedinců. Ve fázi organizace je vybrán lídr a migrant. Aby byla migrace smysluplná, měl by lídr představovat nejlepšího a migrant naopak nejhoršího jedince z populace. V této fázi je uplatněn Paretův princip, jež by se dal shrnout tak, že 20% jedinců nese 80% informace populace. Je tedy jasné, že v horních 20% populace se musí nacházet lídr a v ostatních 80% migrant. Z těchto dvou skupin je následně opět

vytřízeno 20% nejlepších jedinců a z nich jsou vybráni lídr a migrant. Dělení skupin zobrazuje Obrázek 3. Narozdíl od klasické SOMA se zde jedinci pohybují s proměnným *PRT* faktorem.



Obrázek 3: Postup výběru lídra a migranta v SOMA Pareto

Také parametr *Step* je proměnlivý, jen počet skoků v jedné migrační fázi migranta je fixní.

S novou strategií také přichází Tomaszek [12]. Jedná se o strategii AllToNBest, kdy lídr je vybrán náhodně, avšak pouze z N nejlepších jedinců v populaci.

Zajímavou variantou je jistě také hybridní SOMA algoritmus [4, 5] kombinující SOMA s kulturním algoritmem [6], ten zajišťuje uchovávání informace, kde se nachází intervaly tzv. nejvhodnějších řešení. Na základě toho pak dochází k úpravě pohybu jedinců. Hybridního přístupu pak bylo využito také v [13, 14], kde byla zkombinována SOMA s genetickými algoritmy [15].

Pokud se jedná o víceúčelovou optimalizaci, pak v této oblasti byla v roce 2011 představena varianta MOSOMA, jež uvedli Kadlec a Raida [16, 17]. Tato metoda je využitelná jak pro ohraničené, tak pro neohraničené víceúčelové optimalizační problémy a stejně tak jak v reálném číselném oboru, tak v diskrétním.

SOMA algoritmus nemusí nutně sloužit pouze pro použití v oboru reálných čísel. V roce 2013 byla uvedena DSOMA učena pro řešení permutačních kombinatorických problémů [9]. Tuto metodu představil Davendra. Pohyb jedince je zde realizován po skocích v celočíselném oboru.

3 Algoritmus pro optimalizaci hejnem částic

Algoritmus pro optimalizaci hejnem částic (PSO) je algoritmus, který se řadí mezi hejnové algoritmy. Algoritmus byl představen v roce 1995 a jeho autoři jsou Russell Eberhart a James Kennedy [18]. Jeho průběh byl inspirován chováním hejnových druhů zvířat, například ptáky či rybami při hledání potravy, kdy se jedinci shlukují k objektu zájmu, aniž by docházelo k jejich vzájemnému narážení jeden do druhého. Tak jako jemu příbuzné algoritmy se zabývá řešením optimalizačních problémů na multimodálních, nekonvexních, nediferencovatelných a nespojitých funkcí.

3.1 Obecné principy PSO

Tak jako i jiné populační stochastické optimalizační techniky také PSO algoritmus je inicializován na začátku populací nahodilých jedinců v prohledávaném prostoru možných řešení. Poté dochází k cyklickému prohledávání prostoru na základě měnící se rychlosti a polohy částice, dokud nejsou splněny ukončující podmínky algoritmu. Každá částice představuje potenciální řešení daného optimalizačního problému a mění svůj směr pohybu na základě svého vlastního zkoumání a také na základě nejlepšího výsledků celé skupiny. Výpočet rychlosti částice a její nové rychlosti pak charakterizují vztahy (5) a (6).

$$\vec{v}_i^{it+1} = w \cdot \vec{v}_i^{it} + c_1 \cdot rand_1 \cdot (\vec{pBest}_i^{it} - \vec{x}_i^{it}) + c_2 \cdot rand_2 \cdot (\vec{gBest} - \vec{x}_i^{it}), \tag{5}$$

$$\vec{x}_i^{it+1} = \vec{x}_i^{it} + \vec{v}_i^{it}, \tag{6}$$

kde it prezentuje iteraci, i značí pořadí částice v hejnu, $\vec{x}_i^{it}$ představuje pozici částice v daném kroku, $\vec{v}_i^{it}$ značí rychlost částice v daném kroku a $\vec{v}_i^{it+1}$ pak rychlost částice v následujícím kroku. Nejlepší poloha částice s indexem i je pak uložena v proměnné $\vec{pBest}_i^{it}$ a nejlepší poloha celého hejna je reprezentována hodnotou $\vec{gBest}$. $rand_1$ a $rand_2$ jsou pak náhodně generované hodnoty z rovnoměrného rozdělení v rozmezí [0, 1] a c_1 a c_2 jsou učící faktory. Jak již bylo řečeno, pro kontrolu rovnováhy mezi explorací a exploitací je použit parametr setrvačnosti w. Populace je tvořena částicemi, proto bylo voleno stejné značení pro jedince, a to x.

Chod algoritmu by se pak dal specifikovat následujícím pseudokódem:

it - pomocná proměnná představující jednotlivé iterace
y_i - částice v hejně s indexem i
Vstupy:
$iterace$ - maximální počet iterací
f_{obj} - účelová funkce
N - počet částic v hejně
$gBest$ - nejlepší nalezené řešení v celém hejnu

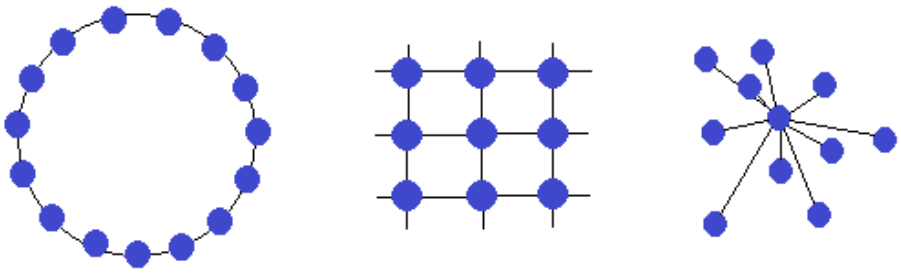
$pBest$ - nejlepší nalezené řešení pro danou částici
for $i \leq N$ **do**
  $p_i = rand_i$
end for
for $it < iterace$ **do**
  for $i \leq N$ **do**
    aktualizuj rychlost částice y_i dle (5)
    aktualizuj polohu částice y_i dle (6)
    $f_{obj}(y_i)$ - ohodnocení pomocí účelové funkce
    if $f_{obj}(y_i) < f_{obj}(pBest)$ **then**
      $pBest = y_i$
    end if
    if $f_{obj}(y_i) < f_{obj}(gBest)$ **then**
      $gBest = y_i$
    end if
  end for
end for

3.2 Parametry PSO

PSO se řadí taktéž mezi lehce implementovatelné algoritmy s velmi dobrými výsledky a rychlou konvergenci k optimu. A protože si celé hejno uchovává výsledek nejlepší částice, může lehce dojít k tomu, že se kolem ní začnou ostatní částice hromadit, čímž hejno ztratí svou různorodost. Pokud se pak tato nejlepší částice nachází v blízkosti některého z lokálních extrému, může dojít ke stagnaci algoritmu a uvíznutí hejna v tomto lokálním extrému. K eliminaci tohoto jevu přispěl parametr setrvačnosti w, který v roce 1998 představili Shi a Eberhart [19]. V případě vysokých hodnot setrvačnosti dochází k hledání globálních extrémů a naopak pokud je parametr nastaven na nižší hodnoty, algoritmus má tendenci uvíznout předčasně v některém z lokálních extrémů. V roce 1999 představili Clerc a Kennedy další parametr zvaný konstrikční faktor, jež byl využit místo setrvačnosti [20]. Parametry ovlivňující chod algoritmu jsou shrnuty v Tabulce 3.

Tabulka 3: Nastavitelné parametry PSO

Parametr	Popis
dimenze D	počet parametrů vektoru, určeno účelovou funkcí
rozsah	udává velikost prohledávaného prostoru
počet částic	obvykle platí pravidlo, že jeho hodnota je $10D$ (obvykle 20 až 40)
v_{max}	maximální rychlost částice, doporučuje se 1/20 rozsahu
učící faktor c_1	udává tendenci vracet se k nejlepší pozici částice
učící faktor c_2	udává tendenci vracet se k pozici nejlepší částice
setrvačnost w	postupně degraduje rychlost částice, doporučená hodnota 0,4 až 0,9



Obrázek 4: Topologie sousedství: kruh, Von Neumannovo, hvězda

3.3 Varianty PSO

Jelikož je PSO algoritmus, který je možné snadno implementovat a také integrovat s jinými optimalizačními technikami, vzniklo více verzí ve snaze o zlepšení jeho výsledků.

3.3.1 Sousedství

Tato varianta vznikla brzy po základní variantě algoritmu s cílem redukovat situace, kdy dojde k uvíznutí v lokálním extrému účelové funkce. Částice jsou rozděleny do tzv. sousedství, což si lze představit jako jakousi formaci či skupinu [21]. Tyto mohou být dvojího typu - *geografické* nebo *sociální*. Geografické sousedství označuje uspořádání částic, kdy se částice nacházejí v jedné oblasti. Zde se však mohou vyskytnout komplikace ve formě určení hranic jednotlivých oblastí a následně určení, zda částice do konkrétní oblasti ještě patří nebo je již součástí jiné. Nejčastěji používanou formou sousedství je sociální, zde je příslušnost částice ke skupině řízená jejím identifikátorem, nikoliv její polohou. V této variantě PSO je pak použit další parametr udávající počet částic v jednom sousedství. Tento parametr však nemá zásadní vliv na výsledky. Nejčastější topologie sousedství jsou vyobrazeny na Obrázku 4. Topologie, která je využívána nejčastěji, je topologie kruhu. Jednotlivé částice jsou v sousedství vždy s částicí, která je k nim nejblíže svým indexem. Jednotlivé skupiny tedy nejsou striktně odděleny, nýbrž se vzájemně prolínají. Z tohoto důvodu dochází k vlivu každé částice na všechny ostatní, ať už přímo, či nepřímo. V případě sousedství je pozměněn výpočet rychlosti částic tak, že místo globálně nejlepší částice je ve výpočtu využita nejlepší částice z téhož sousedství.

INPSO. Neboli PSO nezávislých sousedství a jak již název napovídá, v této variantě jsou k výpočtu využívána sousedství, která na sobě navzájem nezávisí [1].

PSO dynamických sousedství. U této varinaty částice přecházejí mezi sousedstvími, do jejichž blízkosti se dostanou [1].

3.3.2 PSO dle setrvačnosti

Jak již bylo zmíněno, hodnota setrvačnosti w v algoritmu hraje velkou roli, protože ovlivňuje poměr mezi exploitací (lokální prohledávání) a explorací (globální prohledávání). Toto má vliv na celkové výsledky algoritmu. Proto je tomuto parametru věnována samostatná následující část práce, neboť v průběhu let docházelo ke stále novým poznatkům, a tedy i vývoji setrvačnosti w [21, 22].

Lineární setrvačnost. Je známé, že poměr mezi lokálním a globálním prohledáváním má velký dopad na úspěšnost algoritmu. V počátcích výzkumů byla setrvačnost nastavena jako konstanta, avšak v průběhu let bylo zjištěno, že tento přístup není nejoptimálnější a lze tak jen těžko korigovat obě prohledávací fáze. Bylo také zjištěno, že vysoké hodnoty setrvačnosti jsou ku prospěchu globálního prohledávání, avšak naopak nižší hodnoty vedou k lokálnímu prohledávání. Proto byla představena setrvačnost klesající lineárně na základě iterační fáze, ve které se algoritmus nachází [22]. Výpočet pak probíhá dle (7).

$$w^{it} = \frac{it_{max} - it}{it_{max}}(w_{max} - w_{min}) + w_{min}, \tag{7}$$

kde it značí iteraci, ve které se momentálně algoritmus nachází, it_{max} je maximální počet iterací, pro jaký chceme nechat algoritmus běžet, w_{max} představuje maximální přípustnou hodnotu setrvačnosti a w_{min} naopak nejnižší možnou přípustnou hodnotu setrvačnosti.

Nelineární setrvačnost. Inspirujíce se lineárně klesajícím výpočtem setrvačnosti představil G. Chen nelineární setrvačnost [23]. Taková má průběh vždy po určité křivce - například parabola otvírající se nahoru či dolů, exponenciální funkce či sigmoida. Experimenty ukázaly, že při jejich použití dochází k rychlejší konvergenci algoritmu než při využití lineární setrvačnosti.

Setrvačnost na základě fuzzy logiky. Jak šel výzkum v této oblasti kupředu, začaly se také objevovat různé přístupy k výpočtu setrvačnosti tak, aby docházelo ke zlepšení výsledků PSO. Dalším přístupem v této oblasti je výpočet setrvačnosti na základě fuzzy logiky [24]. Nejprve byl představen postup, při kterém byl ze dvou vstupů generován jeden výstup. Jako vstupní parametry sloužily nejlepší globální výsledek a aktuální hodnota setrvačnosti, přičemž výstupem je nová hodnota setrvačnosti. Později byla vyvinuta metoda, která ze dvou vstupů vrací dva výstupy.

Náhodně volená setrvačnost. S přihlédnutím k tomu, jak dynamicky funguje reálný svět, došlo k představení setrvačnosti počítané stochastickým přístupem dle (8).

$$w = 0,5 + \frac{rand}{2}, \tag{8}$$

kde $rand$ představuje náhodnou hodnotu z rovnoměrného rozdělení na intervalu [0,1]. V reálném dynamicky fungujícím systému je velmi těžké predikovat, co bude v daném čase výhodnější, zda globální či lokální prohledávání a právě tento problém se snaží reflektovat tento přístup výpočtu setrvačnosti [21].

Konstrikční faktor. V roce 1999 Clerc představil parametr χ s názvem konstrikční faktor [20, 22], který je používán místo setrvačnosti w. Rychlost je pak počítána dle (9).

$$\vec{v}_i^{it+1} = \chi[\vec{v}_i^{it} + c_1 \cdot rand1 \cdot (p\vec{B}est_i - \vec{x}_i^{it}) + c_2 \cdot rand2 \cdot (g\vec{B}est - \vec{x}_i^{it})], \tag{9}$$

kde konstrikční faktor je počítán dle (10) a (11).

$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}, \tag{10}$$

$$\varphi = c_1 + c_2. \tag{11}$$

Doporučené hodnoty jsou $\chi = 0,729$, $c_1 = c_2 = 2,05$ a $\varphi = 4,1$.

3.3.3 Hybridní PSO

Kombinování jiného optimalizačního algoritmu s PSO bývá aplikováno ze dvou důvodů. Prvním je, aby se zvýšila různorodost částic a tím se zamezilo předčasné konvergenci. Druhým je zlepšení lokálního prohledávání PSO [25].

Modely, které šly tímto směrem, pak do PSO uvedly řadu genetických operátorů jako např. výběr, což představili ve svých pracech Angeline v roce 1998 [26] a Lovbjerg v roce 2001 [27]. Dále bylo využito křížení, čímž se ve svých pracích zabývali Angeline [28] a také Chen [29], dále pak Tsafarakis [30] experimentoval se zavedením operátoru mutace. Cíle byly vždy shodné, a to, aby se algoritmus dokázal vymanit z lokálního extrému. V roce 2015 pak Meng [31] představil CSPSO, jež se lišilo od ostatních tím, že částice v hejnu byly vyjádřeny přímo svou dosud nejlepší nalezenou polohou. V Roce 2009 Vlachogiannis a Lee [32] představili PSO s vylepšenou řídicí rovnicí tak, aby docházelo k lepší komunikaci mezi jednotlivými částicemi a bylo tak posíleno lokální prohledávání. Částice byly schopny reagovat jak na svou doposud nejlepší nalezenou pozici, tak na nejlepší nalezenou pozici ostatních částic namísto globálně nejlepší nalezené pozice. Selvakumar a Thaunushkodi pak v roce 2009 [33] představili CSO, jež je kombinací civilizačního algoritmu a PSO opět proto, aby došlo ke zlepšení komunikace mezi částicemi. Algoritmus využívá shlukového prohledávání. V roce 2014 pak Shin a Kita [34] využili při výpočtech druhé globálně nejlepší nalezené polohy částice a druhé nejlepší polohy jednotlivých částic, čímž se výkonnost algoritmu zvýšila. Další význačná varianta je spojením dvou algoritmů, a to INPSO a GPEA. Přednost INPSO je, že rychle konverguje a výhodou GPEA zase, že je vždy úspěšný v hledání optima a právě obě tyto vlastnosti jsou v tomto druhu PSO využity [1]. Modifikací v tomto směru vznikalo velké množství, uvedené varianty jsou spíše výčtového charakteru pro představu čtenáře, že výzkum je na poli optimalizačních algoritmů bohatý a stále přibývají nové poznatky a vylepšení.

4 Diferenciální evoluce

Diferenciální evoluce (DE) je jednou z nejúspěšnějších metod optimalizace spojitých funkcí. Tato metoda vznikla v roce 1995 na základě výzkumu Kenneta Pricea a Rainera Storna při řešení Čebyševovy polynomiální aproximace [35]. Diferenciální evoluce ke svému běhu využívá pouze malé množství kontrolních parametrů, je robustní, jednoduchá na použití a dá se velmi dobře paralelizovat. Od svého představení byla DE a její varianty použita k řešení velkého počtu reálných problémů napříč různými doménami vědy a techniky [36, 37, 38, 39].

4.1 Principy DE

Diferenciální evoluce začíná s náhodně generovanou počáteční populací jedinců a v každém následujícím evolučním cyklu vykonává stejné kroky, přičemž k mutaci bázových vektorů používá rozdílu jedinců ze stejné populace násobeného určitým faktorem. Po průchodu jednotlivými iteračními cykly mají tyto rozdíly tendenci se přizpůsobovat průběhu účelové funkce. Tato automatická adaptace značně zlepšuje vyhledávací schopnosti algoritmu.

Jak již bylo řečeno, DE generuje na svém počátku náhodně generovanou populaci. Jakmile je vytvořena počáteční populace, dojde v každé iteraci k mutaci vytvořením šumového vektoru $\vec{v}_i^g$ pro každého člena populace či cílový vektor $\vec{x}_i^g$. Celý postup algoritmu by se dal pro variantu DE/Rand/1/Bin shrnout následujícím pseudokódem:

Vstupy:
iterace - počet populací
f_{cost} - účelová funkce
F - mutační konstanta
CR - práh křížení
NP - počet jedinců v populaci
D - dimenze problému (počet parametrů jedince)
Prototypový vektor - tzn jakého datového typu budou jednotlivé parametry jedinců
Algoritmus pro hledání minima:
vytvoř náhodnou počáteční populaci jedinců dle prototypového vektoru
každého jedince ohodnoť dle účelové funkce
for $i < iterace$ **do**
  for $j \leq NP$ **do**
    vyber 3 náhodné jedince rozdílné od p_j
    odečti první dva vybrané jedince -> diferenční vektor
    diferenční vektor vynásob F -> váhovaný diferenční vektor
    váhovaný diferenční vektor přičti k třetímu jedinci -> šumový vektor
    for $k \leq D$ **do**
      generuj náhodné číslo x

```
        if x < CR then
          do zkušebního vektoru přiřaď parametr ze šumového vektoru
        else
          do zkušebního vektoru přiřaď parametr z cílového vektoru
        end if
      end for
      ohodnoť zkušební vektor
      if ohodnocení zkušebního vektoru < ohodnocení cílového vektoru then
        do nové populace zařaď zkušební vektor
      else
        do nové populace zařaď cílový vektor
      end if
    end for
    nahraď starou populaci novou
  end for
```

4.2 Fáze DE

V následující části práce bude popsáno, jakými fázemi algoritmus prochází a tyto fáze budou detailněji popsány a rozepsány.

První iterace algoritmu se skládá z celkem 4 fází:

- inicializace
- mutace
- křížení
- výběr

Přičemž první z nich probíhá pouze pro první generaci a poslední 3 zmíněné se pak iteračně opakují pro všechny generace až do splnění ukončujícího kritéria. Tím je např. daný počet ohodnocení účelovou funkcí nebo počet generací.

4.2.1 Inicializace

Jak již bylo řečeno v předchozí části této práce, DE jako evoluční algoritmus pracuje s populacemi jedinců. Na začátku je tedy na zkoumané oblasti generována náhodná počáteční populace P^0 o velikosti k ($k \geq 4$). Každý člen populace neboli jedinec je D-dimenzionální vektor reálných čísel, přičemž každý člen vektoru $x_{i,j}$ je náhodně generován v rozmezí $[x_{min,j}, x_{max,j}]$, což reprezentuje dolní a horní hranici pro každý člen, který následně vypočteme dle (2). Každý jedinec představuje možné řešení problému. Poté, co je jedinec vygenerován, je ohodnocen pomocí účelové funkce $f_{obj}()$.

4.2.2 Mutace

Další fází, která následuje po inicializaci, je mutace. DE k mutaci využívá kromě cílového vektoru také proměnlivý počet dalších vektorů populace závisející na verzi DE. K mutaci je využita také mutační konstanta F. Základní verze DE pak mutaci provádí dle (12).

$$\text{DE/rand/1: } \vec{v}_i^g = \vec{x}_{R_1^i}^g + F \cdot (\vec{x}_{R_2^i}^g - \vec{x}_{R_3^i}^g), \tag{12}$$

kde $\vec{v}_i^g$ značí výsledný šumový vektor, indexy R_1^i, R_2^i, R_3^i jsou postupně různá celá čísla z intervalu $[1, NP]$ a zároveň různá od indexu i (parametr NP značí počet jedinců v populaci). Tato náhodná čísla jsou generována pro každý cílový vektor. Mutační konstanta F je kladný řídicí parametr pro násobení rozdílu vektorů. $\vec{x}_{best}^g$ je nejlepší jedinec v populaci v dané generaci g, tzn. jedinec s nejlepším ohodnocením (v případě minimalizačního problému tedy jedinec s nejnižší hodnotou účelové funkce). Další druhy mutací budou zmíněny v pozdější části této práce.

4.2.3 Křížení

Během křížení dochází ke kombinaci šumového vektoru $\vec{v}_i^g$ s cílovým vektorem $\vec{x}_i^g$, čímž vznikne tzv. zkušební vektor $\vec{u}_i^g$. Nejčastěji používané metody křížení jsou *exponenciální* a *binomické*.

U binomické verze je pro každý parametr cílového vektoru $\vec{x}_i^g$ generováno náhodné číslo v intervalu 0 a 1. Pokud je toto číslo menší nebo rovno řídicímu parametru Cr, který se nazývá také práh křížení, pak je konkrétní parametr cílového vektoru $\vec{x}_i^g$ nahrazen odpovídajícím parametrem ze šumového vektoru $\vec{v}_i^g$. Pokud tato podmínka splněna není, pak k nahrazení nedojde a parametr tedy zůstane neměnný. Celý postup se dá vyjádřit (13).

$$\vec{u}_{i,j}^g = \begin{cases} \vec{v}_{i,j}^g & \text{když } j = K \text{ nebo } rand_{i,j}[0,1] \leq Cr, \\ \vec{x}_{i,j}^g & \text{jinak,} \end{cases} \tag{13}$$

kde K je náhodně zvolené celé číslo z množiny $\{1, 2, ..., d\}$ a $rand_{i,j}[0,1]$ je náhodné číslo z rovnoměrného rozdělení, které je generováno pro každý parametr vektoru $\vec{x}_i^g$ zvlášť. Číslo K zde zajišťuje, že zkušební vektor $\vec{u}_i^g$ získá alespoň jeden parametr ze šumového vektoru $\vec{v}_i^g$.

U křížení exponenciálního nejprve náhodně vybereme celé číslo n z množiny $\{1, 2, ..., d\}$, které značí počátek v cílovém vektoru $\vec{x}_i^g$, odkud bude prováděno následné křížení. Dále vybereme náhodné číslo L opět z množiny $\{1, 2, ..., d\}$, jež v tomto případě představuje počet odpovídajících parametrů z cílového vektoru $\vec{x}_i^g$ a šumového vektoru $\vec{v}_i^g$, které budou navzájem zaměněny. Tvorba zkušebního vektoru by se pak dala zachytit vztahem (14).

$$\vec{u}_{i,j}^g = \begin{cases} \vec{v}_{i,j}^g & \text{když } j = n \bmod d, (n+1) \bmod d, ..., (n+L-1) \bmod d, \\ \vec{x}_{i,j}^g & \text{jinak.} \end{cases} \tag{14}$$

Čísla n a L jsou generována pro každého jedince zvlášť. Exponenciální křížení je efektivní pouze v případě, že existují konexe mezi jednotlivými parametry vektoru, proto je více využívané křížení binomické.

4.2.4 Výběr

Po fázi křížení následuje fáze selekce neboli výběru. V tomto případě dojde k ohodnocení zkušebního vektoru $\vec{u}_i^g$ a je porovnáno s ohodnocením cílového vektoru $\vec{x}_i^g$. Fáze výběru shrnuje (15).

$$\vec{x}_i^{(g+1)} = \begin{cases} \vec{u}_i^g & \text{když } f_{obj}(\vec{u}_i^g) \leq f_{obj}(\vec{x}_i^g), \\ \vec{x}_i^g & \text{jinak,} \end{cases} \tag{15}$$

kde f_{obj} je účelová funkce. Pokud bychom tedy brali v úvahu minimalizační úlohu, pak by zkušební vektor $\vec{u}_i^g$ vstupoval do nové generaci v případě, že by měl nižší hodnotu účelové funkce než cílový vektor $\vec{x}_i^g$, pokud by tomu tak nebylo, byl by v následující generaci ponechán vektor cílový. Výběr může být synchronní a asynchronní. Během synchronního jsou jedinci ukládáni do nové kolekce, tzn. je udržována jak stará, tak nová generace, avšak během asynchronního jsou jedinci nahrazováni průběžně a noví jedinci se tak již mohou účastnit mutační fáze.

4.3 Parametry DE

Chod diferenciální evoluce je ovlivněn řídicími parametry. Ty jsou spolu s ostatními parametry popsány v Tabulce 4 . Práh křížení CR je vždy reálné číslo z rozmezí [0,1], přičemž, pokud je

Tabulka 4: Řídicí parametry DE

Parametr	Popis	Interval
CR	práh křížení	[0,1]
D	dimenze problému	dána problémem
NP	velikost populace	[$10D$,$100D$]
F	mutační konstanta	[0,2]
$iterace$	udává počet opakování algoritmu	>0

účelová funkce separabilní, pak jsou výhodnější hodnoty u spodní hranice intervalu a v případě neseparabilní účelové funkce jsou naopak výhodnější hodnoty u horní hranice intervalu. Čím vyšší hodnota je za CR volena, tím více parametrů zkušební jedinec převezme od svých rodičů a naopak čím nižší, tím více se bude zkušební jedinec podobat původnímu jedinci a míra mutace bude velmi nízká. Dimenze problému D určuje počet argumentů účelové funkce, což je pevně daná hodnota, která charakterizuje zkoumanou problematiku. Velikost populace NP udává počet jedinců v populaci. Experimentálně bylo zjištěno, že hodnoty bychom měli volit z intervalu [10D, 100D], přičemž nejnižší populace, s níž je algoritmus schopen pracovat, jsou 4 jedinci. Mutační

konstanta F je parametr v intervalu [0, 2], přičemž je doporučeno volit hodnoty v rozmezí 0,3 - 0,9.

4.4 Varianty DE

Následující část textu je věnována druhům DE, neboť v této oblasti proběhlo a stále probíhá velké množství experimentů a pokusů o zlepšení tohoto algoritmu.

4.4.1 Varianty dle druhu mutace

Tyto verze jsou založeny na rozdílném výpočtu šumového vektoru a jejich názvy z těchto výpočtů vycházejí. Jmenná konvence vychází z následujícího pravidla DE/α/β/ω, kde DE značí, že se jedná o diferenciální evoluci, α značí, který vektor je perturbovaný, β je pak počet rozdílů dalších vektoru různých od α, které budou využity k perturbaci a ω potom symbolizuje druh křížení, který byl využit (například *exp* - exponenciální, *bin* - binomické). Varianty jsou shrnuty v (16a) - (16e).

$$\textbf{DE/rand/1/exp: } \vec{v}_i^g = \vec{x}_{R_1^i}^g + F \cdot (\vec{x}_{R_2^i}^g - \vec{x}_{R_3^i}^g), \tag{16a}$$

$$\textbf{DE/best/1/exp: } \vec{v}_i^g = \vec{x}_{best}^g + F \cdot (\vec{x}_{R_1^i}^g - \vec{x}_{R_2^i}^g), \tag{16b}$$

$$\textbf{DE/current-to-best/1/exp: } \vec{v}_i^g = \vec{x}_i^g + F \cdot (\vec{x}_{best}^g - \vec{x}_i^g) + F \cdot (\vec{x}_{R_1^i}^g - \vec{x}_{R_2^i}^g), \tag{16c}$$

$$\textbf{DE/best/2/exp: } \vec{v}_i^g = \vec{x}_{best}^g + F \cdot (\vec{x}_{R_1^i}^g - \vec{x}_{R_2^i}^g) + F \cdot (\vec{x}_{R_3^i}^g - \vec{x}_{R_4^i}^g), \tag{16d}$$

$$\textbf{DE/rand/2/bin: } \vec{v}_i^g = \vec{x}_{R_1^i}^g + F \cdot (\vec{x}_{R_2^i}^g - \vec{x}_{R_3^i}^g) + F \cdot (\vec{x}_{R_4^i}^g - \vec{x}_{R_5^i}^g), \tag{16e}$$

kde indexy $R_1^i, R_2^i, R_3^i, R_4^i, R_5^i$ jsou postupně různá celá čísla z intervalu $[1, NP]$ a zároveň různá od indexu i. Tato náhodná čísla jsou náhodně generována pro každý původní vektor. Mutační konstanta F je kladný řídicí parametr pro násobení rozdílu vektorů. $\vec{x}_{best}^g$ je nejlepší jedinec v populaci v daném iteračním cyklu g, tzn. jedinec s nejlepším ohodnocením (v případě minimalizačního problému tedy jedinec s nejnižší hodnotou účelové funkce).

4.4.2 Další varianty DE

Výkonnost DE je úzce spjata s nastavením řídicích parametrů F a CR, jež jsou však závislé na řešeném problému a mohou tak být pro každou účelovou funkci jiné. Toto nastavování může být pracné a velmi náročné na prostředky. Proto bylo za účelem zjištění, které hodnoty řídicích parametrů jsou nejlepší pro daný problém, provedeno velké množství experimentů. Později pak byly představeny metodiky, při nichž algoritmus tyto parametry sám přizpůsobuje aktuálnímu stavu výpočtu. Z tohoto vychází např. varianta SaDE (z anglického Self Adaptive DE) [40]. Zde dochází k přizpůsobování mutační konstanty F a prahu křížení Cr na základě historické úspěšnosti generovaného zkušebního vektoru.

V roce 2006 Brest a kolektiv představili jDE variantu DE [41], jež k adaptaci řídicích parametrů (F a CR) využívá evoluční proces jedinců. Dimenze jedinců je zvýšená o 2 a oba řídicí parametry (F a CR) jsou do jedince umístěny spolu s jeho ostatními parametry. Poslední dva parametry neprochází klasickými operacemi mutace a křížení a jsou přepočítávány zvlášť.

V roce 2007 pak Zhang a Sanderson představili variantu JADE [42], která přinesla novou mutační strategii „DE/current-to-p-best“. Tato varianta využívá adaptivní přístup v nastavování řídicích parametrů s externím archivem. Neuchovává pouze nejlepší nalezené řešení, avšak také množinu řešení, která lze stále ještě považovat za dobrá. Tato dobrá řešení mohou poskytnout dodatečnou informaci o směru, kterým se má hledání ubírat.

V roce 2013 byla uvedena jiná varianta adaptivní DE, a to SHADE (z anglického „Success-History Based Parameter Adaptation for Differential Evolution“). Varianta vychází z varianty JADE a představili ji Tanabe a Fukunaga [40]. Stejně jako JADE i SHADE využívá mutační strategii „DE/current-to-p-best“. Největším přínosem SHADE oproti JADE algoritmu je v jeho adaptivním nastavování parametrů F a CR na základě historicky dobrých hodnot, jež jsou uchovávány.

Poikolainen a kolektiv zase v roce 2015 navrhli techniku tvorby populace inspirovanou shlukováním (CBPI) [43]. Inicializace je rozdělena do tří fází. V první fázi dochází k lokálnímu prohledávání na náhodné kolekci rovnoměrně rozmístěných bodů. V druhé fázi je na nalezenou populaci aplikováno shlukování pomocí *k-means* algoritmu a ve třetí fázi jsou z vytvořených shluků vybráni nejlepší jedinci, kteří budou součástí populace.

Vědeckých výzkumů s cílem vylepšit algoritmus od doby jeho představení proběhlo hodně, v předchozí části jsou nastíněny pouze některé z nich. Přehled dalších variant je k nalezení například v [37].

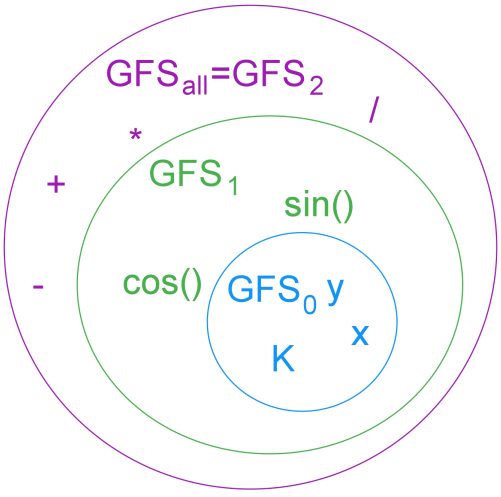
5 Symbolická regrese

Symbolická regrese je proces, který skládá malé části do větších celků. Může být použita například k vhodnému sestavení logických obvodů, jejichž výstupem je tabulka pravdivostních hodnot, avšak může také sestavovat matematickou rovnici, jež popisuje naměřená data a jejich vzájemné závislosti. Nejznámějšími metodami symbolické regrese jsou například analytické programování [1, 44], genetické programováním [45] nebo gramatická evoluce [46]. Další části budou věnovány právě analytickému programování, které bylo v této práci použito.

5.1 Analytické programování

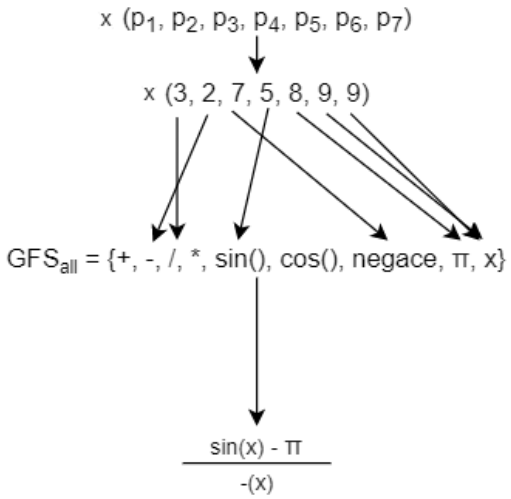
Tato metoda je založena principu DSH [1, 44]. Velmi zjednodušeně by se dalo říci, že se jedná o zobrazení z množiny symbolů do množiny možných řešení. Jednotlivé parametry jedince, jež se účastní evolučního cyklu, netvoří přímo jednotlivé stavební prvky, avšak pouze odkazy do množiny symbolů. Tyto odkazy jsou ve formě indexů jednotlivých prvků. Pro účely tohoto odkazování je vytvořena množina GFS_{all} tvořená všemi symboly, ze kterých jedinec může být složen. Mohou to být například funkce, konstanty, operátory, obecně tedy terminály a neterminály. Tato množina je uspořádaná dle počtu parametrů, jež daný prvek přijímá. GFS_{all} je tedy sjednocením jednotlivých GFS_k, kde index k prezentuje množství přijímaných parametrů.

$GFS_{all} = GFS_2 \cup GFS_1 \cup GFS_0$
$GFS_2 = \{+, -, /, *, sin(), cos(), K, x, y\}$
$GFS_1 = \{sin(), cos(), K, x, y\}$
$GFS_0 = \{K, x, y\}$
Jednotlivé množiny jsou pro lepší ilustraci zobrazeny na Obrázku 5.



Obrázek 5: Zobrazení množin symbolů a jejich vzájemné vztahy

Evolučního cyklu se tedy účastní jedinec složený z indexů, avšak účelovou funkcí je hodnocen již výraz vzniklý překladem indexů. Sestavování rovnice pak probíhá dle Obrázku 6.



Obrázek 6: Schéma tvorby výrazu z jedince

Jednotlivé fáze tvorby výrazu na obrázku se dají shrnout následovně:

1) Máme jedince x, jež má parametry $p_1, p_2, p_3, p_4, p_5, p_6, p_7$. Hodnoty parametrů neboli indexy do množiny GFS_{all} jsou pak po řadě $(3, 2, 7, 5, 8, 9, 9)$. Vezmeme tedy první parametr jedince p_1, jež má hodnotu 3 a promítneme jej do množiny GFS_{all}, dostáváme funkci podíl, jež má dva parametry.

2) První z parametrů podílu obsadíme další funkcí, jež odpovídá druhému parametru jedince p_2, a tedy indexu 2 v množině GFS_{all}, zde se nachází funkce minus, jež má také dva parametry. Ty však budeme doplňovat později.

3) Nyní jako druhý parametr podílu vezmeme další parametr jedince p_3, jež odpovídá negaci výrazu. Negace má jeden parametr, ten budeme doplňovat také později, jakmile na něj přijde řada.

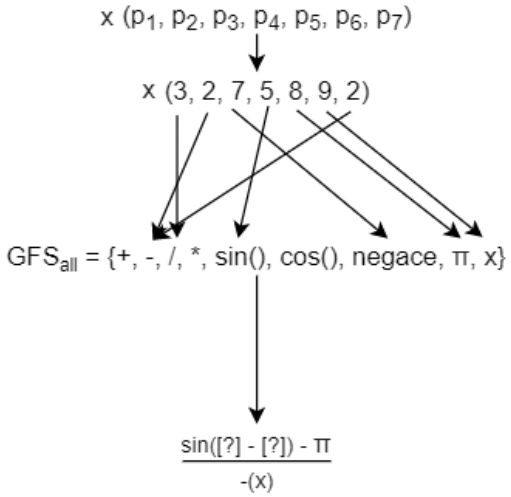
4) Jako další je třeba dosadit první parametr funkce minus. Parametr v jedinci, jež následuje, je p_4 a má hodnotu 5. Na tomto indexu leží v GFS_{all} funkce $sin()$, jež má opět parametr, který doplníme později.

5) Nyní následuje druhý parametr funkce minus. Vezmeme tedy další parametr jedince, a to p_5. Tento má hodnotu 8. Pod indexem 8 v množině GFS_{all} leží konstanta π.

6) Jako další krok je potřeba doplnit parametr do funkce negace. Opět tedy vyčteme v pořadí další parametr jedince p_6, jehož hodnota je 9 a odkazuje tedy do GFS_{all} na bezparametrickou konstantu x.

7) Posledním krokem je doplnění parametru do funkce sin(). V jedinci tedy zbývá poslední parametr p_7 s hodnotou 9, jež odkazuje opět na konstantu x. Tímto je celý výraz uzavřen a validní.

V tomto případě by byla tvorba výrazu triviální. Může však nastat situace, kdy by indexy do GFS_{all} vedly k tvorbě patologického jedince - tedy nevalidního výrazu. Takový případ je ilustrován na Obrázku 7.



Obrázek 7: Schéma tvorby výrazu z patologického jedince

V tomto případě by prvních 6 kroků tvorby výrazu bylo stejných jako v předešlém příkladě na Obrázku 6. Sedmý krok by se však lišil, neboť v okamžiku, kdy bychom doplnili parametr do funkce sin(), který by odpovídal indexu 2 do množiny GFS_{all}, dostali bychom operaci minus, jež potřebuje jako vstup dva parametry. Ty však nemáme v tomto okamžiku jak obsadit, neboť v jedinci x již nezbývají žádné další parametry, které bychom mohli využít pro další výběr z množiny GFS_k. Výraz, který by tedy vznikl a který vidíme na Obrázku 7, by byl nevalidní. Pokud bychom tedy chtěli konstruovat pouze validní výrazy z jedinců, pak musí dojít v každém kroku k ošetření výběru ze správné množiny GFS_k dle množství indexů zbývajících v jedinci a množství parametrů, které ještě musí být obsazeny. V Kroku 7 tedy je tedy třeba obsadit 1 parametr výrazu a počet zbývajících parametrů v jedinci je 0. Řešením je tedy v takovém případě vybrat symbol z množiny s počtem parametrů 0, a tedy z množiny terminálů. Jedná se o druhý prvek z množiny GFS_0 tedy x.

AP není samo o sobě samostatnou evoluční technikou, ale spíše způsobem zobrazení, potažmo způsobem tvorby výrazu. Úspěšnost AP tedy závisí na zvoleném optimalizačním algoritmu, jenž bude nad jednotlivými jedinci spuštěn, a na jeho úspěšnosti.

Modifikací AP je varianta zvaná *Posílené hledání*. Princip této varianty spočívá v tom, že nejlepší nalezený výraz (často v podobě rovnice) je přidán do množiny GFS_0 nakonec jako bezparametrický terminál. Pokud je v průběhu vykonávání evolučního algoritmu nalezen výraz, který má lepší ohodnocení, pak je tento výraz vyměněn za výraz již přidaný do GFS_0 [7].

6 Implementace

Jelikož předchozí části byly věnovány teoretické stránce této práce, v následujících kapitolách bude přiblížena praktická implementace řešení.

6.1 Použité nástroje

Řešení je implementováno v jazyce *C++* ve verzi *C++17* a je cíleno pro operační systémy s jádrem Linux.

Ke kompilaci kódu byl vybrán kompilátor *GCC* ve verzi 8.3.1. Byla testována také kompatibilita s kompilátorem *Clang* ve verzi 7.0.1. Pro sestavení programu byl pak zvolen nástroj *CMake* ve verzi 3.14.5 pro jeho velmi snadné a uživatelsky přívětivé použití a funkce zcela vyhovující této práci. Kromě standardní C++ knihovny byla použita také knihovna *Boost* ve verzi 1.66.

Pro samotnou implementaci bylo využito vývojové prostředí *Qt Creator* pro jeho dostupnost a také proto, že se jedná o velice kvalitní IDE pro jazyk *C++*. IDE *Qt Creator* také podporuje všechny ostatní zvolené technologie.

Pro přehledné a plně automatické formátování kódu byl zvolen nástroj *ClangFormat*, jelikož sám vykonává rutinní úkony jako je odsazení, údržba délky řádků či jiných velmi důležitých vlastností formátování kódu. Zvolená verze nástroje *ClangFormat* je 7.0.1.

6.2 Struktura programu

Samotné SW řešení je rozděleno do několika logických částí, a to:

- testovací funkce
- metaheuristiky
- nastavení
- pomocné knihovny

Každá logická část je umístěna v samostatném jmenném prostoru, přičemž struktura těchto jmenných prostorů kopíruje adresářovou strukturu projektu. V následujících kapitolách se jednotlivými částmi budu zabývat detailněji.

6.3 Testovací funkce

V této kapitole bude přiblíženo implementační řešení prvního funkčního celku, a to testovacích funkcí, jež byly využity jako účelové funkce jednotlivých metaheuristických algoritmů. Testovací funkce se nachází celé ve jmenném prostoru `cost_functions`. Z důvodu kompatibility byla zvolena cesta vlastní implementace 9 testovacích funkcí uvedených v Tabulce 5. Funkce byly voleny s ohledem na jejich průběhy tak, aby byly zastoupeny jak multimodální, tak unimodální funkce, a byla tak pokryta větší oblast testování metaheuristického algoritmu z více úhlů pohledu.

Tabulka 5: Implementované testovací funkce a jejich modalita

Testovací funkce	Modalita
Rosenbrockova funkce	unimodální
Sférická funkce	unimodální
Zakharova funkce	unimodální
Ackleyho funkce	multimodální
Griewankova funkce	multimodální
Levyho funkce	multimodální
Michalewiczova funkce	multimodální
Rastriginova funkce	multimodální
Schwefelova funkce	multimodální

Byla také vytvořena výčtová třída `cost_functions::Type`, jež slouží jako výčtový seznam jednotlivých testovacích funkcí. Aby byly testovací funkce schopny pracovat s body různých dimenzí, bylo v implementaci voleno řešení šablonových funkcí, které přijímají dimenzi jako jeden ze svých argumentů. Z tohoto důvodu je veškerý kód (signatury a také implementace metod) umístěn do hlavičkového souboru `testfunctions.hpp`.

Byly vytvořeny také dva zástupné symboly pro zvýšení čitelnosti kódu, jež byly umístěny do samostatných hlavičkových souborů `point.hpp` a `function.hpp` pro lepší přehlednost. Symbol `cost_functions::Point` je zástupce pro třídu `utils::point::Point`, jehož souřadnice jsou typu `double`. Symbol `cost_functions::Function` zastupuje konkrétní specializaci `std::function` používanou pro reprezentaci účelových funkcí napříč celým programem.

V této části kódu se také nachází hlavičkový soubor `factory.hpp` se signaturou a implementací tovární metody `create` pro tvorbu objektu typu `cost_functions::Function` na základě parametru výčtového typu `cost_functions::Type`. Jednotlivé soubory a jejich vztahy jsou znázorněny na diagramu tříd na Obrázku 8.

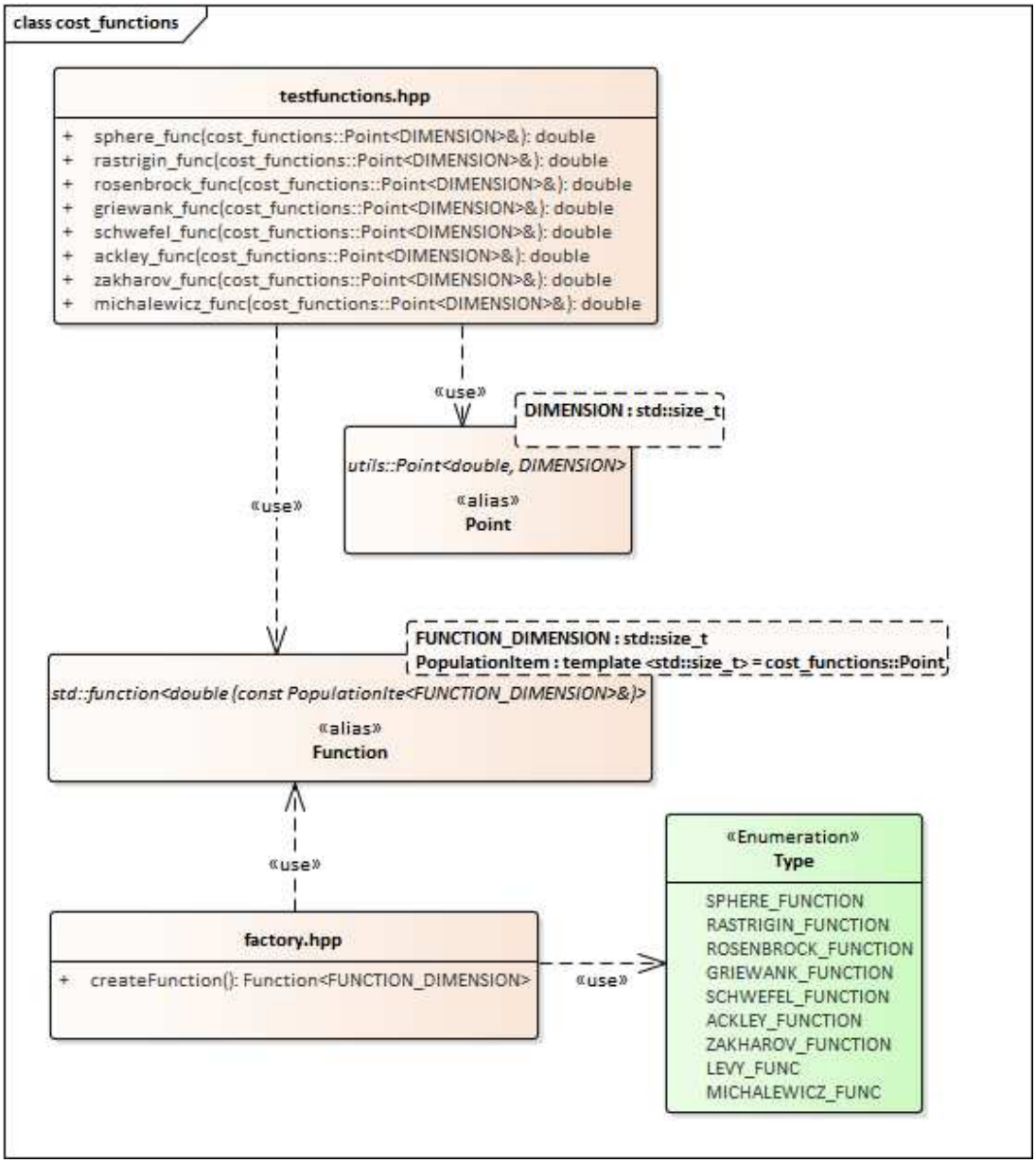
6.4 Metaheuristiky

V této sekci kódu je umístěna nejdůležitější a nejrozsáhlejší část implementace. Nachází se ve jmenném prostoru `metaheuristics` a dělí se na další logické dílčí části:

- Algoritmy
- GFS množiny

Dále také přímo obsahuje hlavičkové soubory `environment.hpp` a `type.hpp`. Struktura této části je pro lepší přehlednost zachycena na diagramu komponent, jež zobrazuje Obrázek 9.

Každá dílčí komponenta či balíček bude dále popsán v samostatné podkapitole.

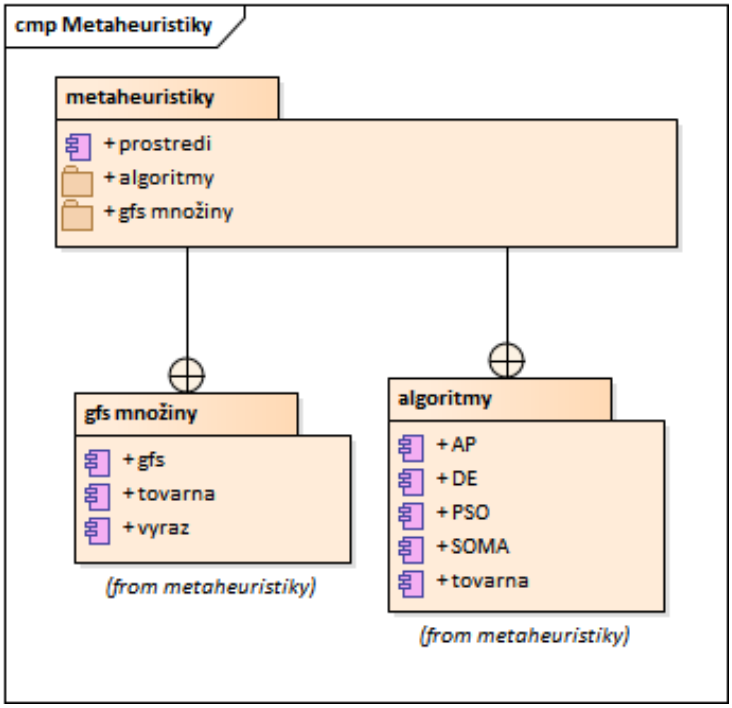


Obrázek 8: Diagram tříd testovacích funkcí

6.4.1 Prostředí

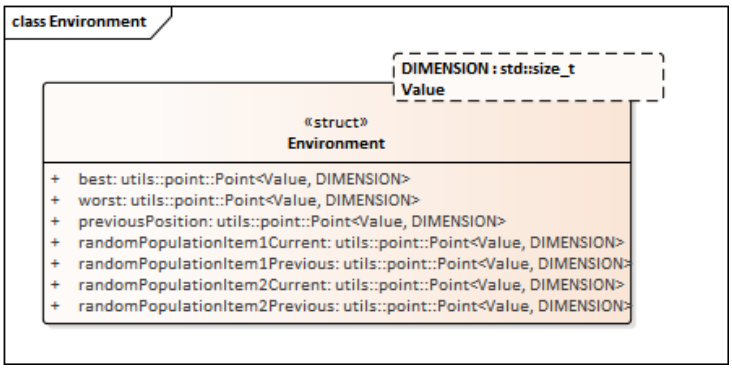
Prostředí je komponenta, jež plní funkci prostředníka mezi metaheuristickým algoritmem sloužícím jako účelová funkce pro AP a mezi AP samotným. Jde o strukturu s názvem `Environment`, jež má dva šablonové parametry `std::size_t DIMENSION` udávající dimenzi jedince metaheuristického algoritmu a parametr `Value` udávající datový typ souřadnic bodů prohledávaného prostoru možných řešení. Vše je pak umístěno v hlavičkovém souboru `environment.hpp`.

Účel komponenty by se dal shrnout následovně. Při tvorbě výrazu v AP jsou využity bezparametrické výrazy z množiny GFS_0, které nabývají hodnot jedinců z populace metaheuristického



Obrázek 9: Diagram komponent a balíčků části metaheuristiky

algoritmu. K těm však AP nemá přímý přístup, jelikož jsou tito jedinci částí populace, jež je součástí vnitřního stavu metaheuristického algoritmu běžícího pod AP. To je důvod, proč jsou tato data předávána ve formě sdíleného ukazatele `std::shared_ptr` na strukturu `Environment`. Struktura `Environment` je vyobrazena na Obrázku 10.



Obrázek 10: Struktura Environment

Sdílený ukazatel `std::shared_ptr` je druh chytrého ukazatele dostupného od *C++11*, který se nachází v knihovně <`memory`>. Tento drží sdílené vlastnictví objektu skrze ukazatel. Je možné, aby více sdílených ukazatelů vlastnilo jeden a ten samý objekt. Objekt je zničen a alokovaná paměť uvolněna právě tehdy, když je zničen poslední sdílený ukazatel, jenž tento objekt vlastní, nebo v případě, že je tomuto poslednímu sdílenému ukazateli přiřazen jiný ukazatel operátorem

operator= nebo metodou reset. Není zde tedy riziko úniku neuvolněné paměti jako u standardních ukazatelů, jelikož C++ nemá tzv. „garbage collector“ jako mají např. programovací jazyky *C#* nebo *Java*.

6.4.2 Algoritmy

Tato část programu je celá umístěna ve jmenném prostoru metaheuristics::algorithms a jsou zde implementace jednotlivých metaheuristických algoritmů. Pro implementaci byli vybráni jejich tři zástupci, a to:

- SOMA
- PSO
- DE

Mimo jiné se v této části programu kromě metaheuristických algoritmů nacházejí také další konstrukce společné pro všechny tři metaheuristické algoritmy. Jsou jimi třída Base, třída Individual, třída Leader, třída Factory a zástupný symbol Population. Všechny konstrukce mají šablonové parametry zajišťující univerzálnost jejich použití. Jejich implementace je tedy celá umístěna v příslušných hlavičkových souborech. Detailům implementace budou věnovány následující části této práce.

Třída Base je bázovou třídou, ze které dědí všechny tři metaheuristické algoritmy a těm poskytuje společné metody pro nastavení a uchování dolní a horní hranice prohledávaného prostoru možných řešení, dále pak uchovává generátor náhodných čísel. Třída má několik šablonových parametrů, z nichž nejdůležitější jsou std::size_t METAHEURISTIC_DIMENSION a tzv. „template template parameter“ PopulationItem. Druhý z nich je nastaven na výchozí hodnotu cost_functions::Point. Tzv. „template template parameter“ je jeden z druhů šablonových parametrů v jazyce C++, který je sám o sobě šablonou. Jeho šablonové parametry je proto potřeba specifikovat při vytvoření objektu daného typu. Tohoto řešení bylo užito s přihlédnutím k pozdější čitelnosti kódu v místě, kde je šablona použita. První šablonový parametr METAHEURISTIC_DIMENSION udává dimenzi problému řešeného metaheuristickým algoritmem. Třída má parametrický konstruktor, jež přijímá dva parametry - bod na dolní hranici optimalizovaného prostoru borderMin a bod na horní hranici optimalizovaného prostoru borderMax, které jsou uloženy do chráněných proměnných.

Třída Individual byla vytvořena za cílem znázornění jedince v jednotlivých algoritmech. Jedná se opět o šablonu, jež má podobné šablonové parametry jako třída Base. Třída má dva parametrické konstruktory. První z nich přijímá dva parametry - pozici position a účelovou funkci costFunction. Tento konstruktor tedy umožňuje vytvořit jedince přímo na dané pozici v prohledávaném prostoru možných řešení. Druhý konstruktor má čtyři parametry - bod na dolní hranici prohledávaného prostoru border1, bod na horní hranici prohledávaného prostoru border2, generátor náhodných čísel generator a účelovou funkci costFunction. Tento konstruktor volá

pak předchozí a za parametr `position` dosadí náhodně generovanou pozici v prohledávaném prostoru zavoláním metody `getRandomPopulationItem`. Tato metoda je součástí třídy `utils::RngGenerator`. Generátor těchto náhodných pozic se řídí rovnoměrným rozdělením.

Jedinec v sobě uchovává prostřednictvím soukromých proměnných následující údaje:

- svou současnou pozici `m_currentPosition`
- svou předchozí pozici `m_previousPosition`
- svou nejlepší dosud dosaženou pozici `m_bestPosition`
- účelovou funkci `m_costFunction`

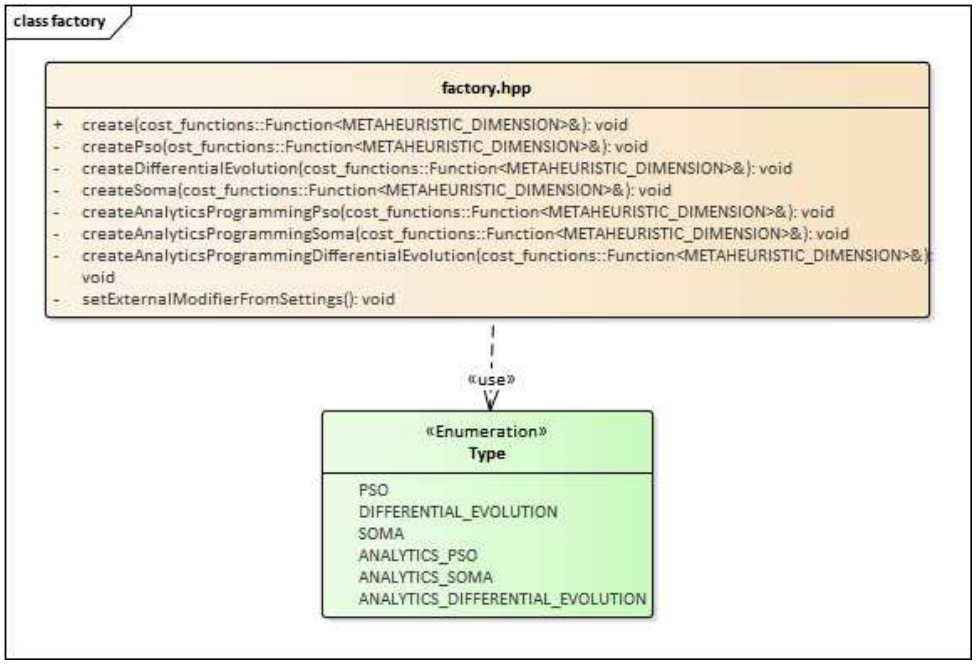
Hodnoty z prvních tří zmíněných soukromých proměnných jsou ve veřejném rozhranní třídy `Individual` přístupné pomocí příslušných `get` funkcí. Jedinci je také možné nastavit novou pozici pomocí metody `setNewPosition`. V této metodě také dochází k potřebným přepočtům - do předchozí pozice jedince se uloží současná pozice, jedincova současná pozice se nahradí novou a pokud je nová pozice lepší než nejlepší dosud nalezená pozice jedince, pak dojde k přenastavení také této hodnoty v jedinci. Celý tento postup je znázorněn v následující ukázce kódu.

```
template <std::size_t DIMENSION, template <std::size_t> typename PopulationItem
    , typename CompareOperation>
void Individual<DIMENSION, PopulationItem, CompareOperation>::setNewPosition(
    const PopulationItem<DIMENSION>& position)
{
    // set previous position
    m_previousPosition = m_currentPosition;
    // calculate the new position's value
    double positionValue = m_costFunction(position);
    // set current position and it's value
    m_currentPosition = {position, positionValue};
    // check if the best position needs to be updated as well
    if (CompareOperation()(positionValue, m_bestPosition.value))
    {
        // update the best position and it's value
        m_bestPosition = {position, positionValue};
    }
}
```

Výpis 1: Ukázka nastavení nové polohy jedince

Třída `Factory` se nachází ve jmenném prostoru `metaheuristics::algorithms` a implementuje funkci `create`. Jelikož má tato funkce dva šablonové parametry – `METAHEURISTIC_DIMENSION`

a `GFS_INDIVIDUAL_DIMENSION` – její signatura včetně její implementace jsou umístěny v hlavičkovém souboru `factory.hpp`. Funkce je šablonovaná z důvodu univerzálního použití pro jakoukoliv dimenzi jedince z množiny GFS. Funkce také umožňuje univerzální použití pro jakoukoliv dimenzi metaheuristického algoritmu sloužícího k ohodnocení daného výrazu AP. Dimenzi jedince AP udává parametr `GFS_INDIVIDUAL_DIMENSION` a dimenzi metaheuristického algoritmu udává parametr `METAHEURISTIC_DIMENSION`. Cílem této funkce je vytvoření instance třídy zastupující metaheuristický algoritmus využitý jako účelovou funkci pro AP na základě parametru výčtového typu `metaheuristics::Type`. Komponenta je zobrazena na třídním diagramu na Obrázku 11.

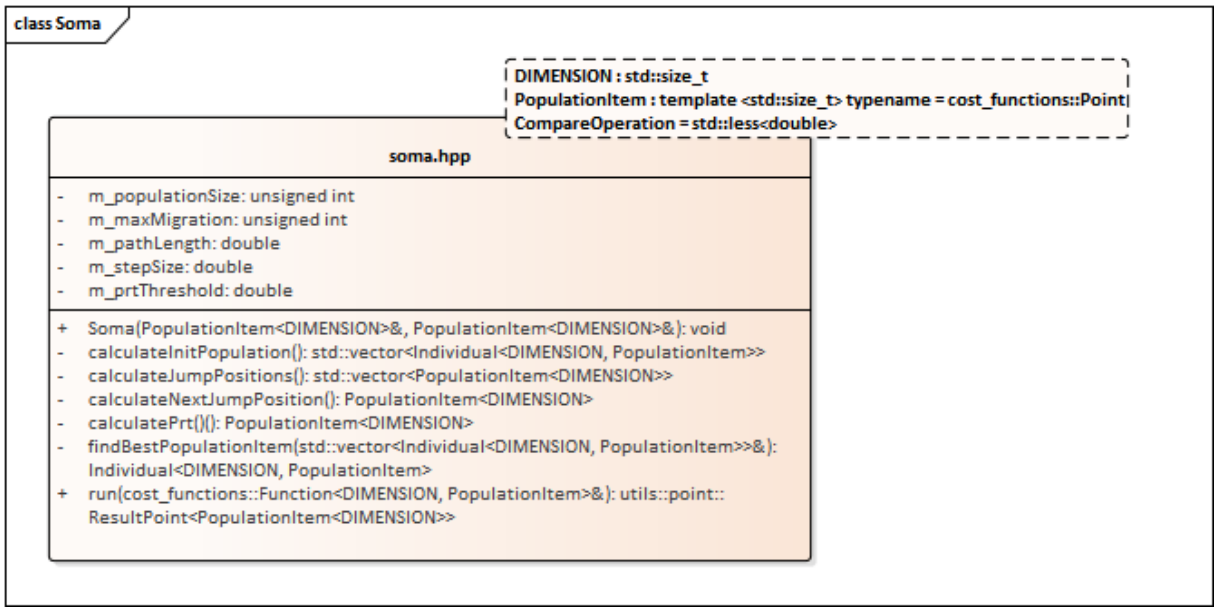


Obrázek 11: Třída továrna - třídní diagram

6.4.3 Implementace algoritmu SOMA

Pro experimentování s metodou symbolické regrese (analytickým programováním) byly voleny celkem tři optimalizační algoritmy a jedním s nich je SOMA. Byla implementována varianta algoritmu *AllToOne*, tedy varianta, kdy je stanoven lídr a ostatní jedinci k němu po řadě putují.

Algoritmus byl implementován formou šablonované třídy z toho důvodu, aby byl schopen pracovat jak s jedincem tvořeným indexy odkazujícími do množiny GFS_{all} v analytickém programování, tak s jedincem, jenž představuje bod v multidimenzionálním prostoru jednotlivých testovacích funkcích. Celý zdrojový kód je tedy z tohoto důvodu umístěn v hlavičkovém souboru `soma.hpp`. Třída dědí z Báze (`base.hpp`), proto implementuje také všechny její metody pro práci s výrazem, sestaveným pomocí AP, aby algoritmus mohl být využit jako účelová funkce pro AP. Metody, jež šablona implementuje, jsou uvedeny na Obrázku 12.



Obrázek 12: Šablona Soma

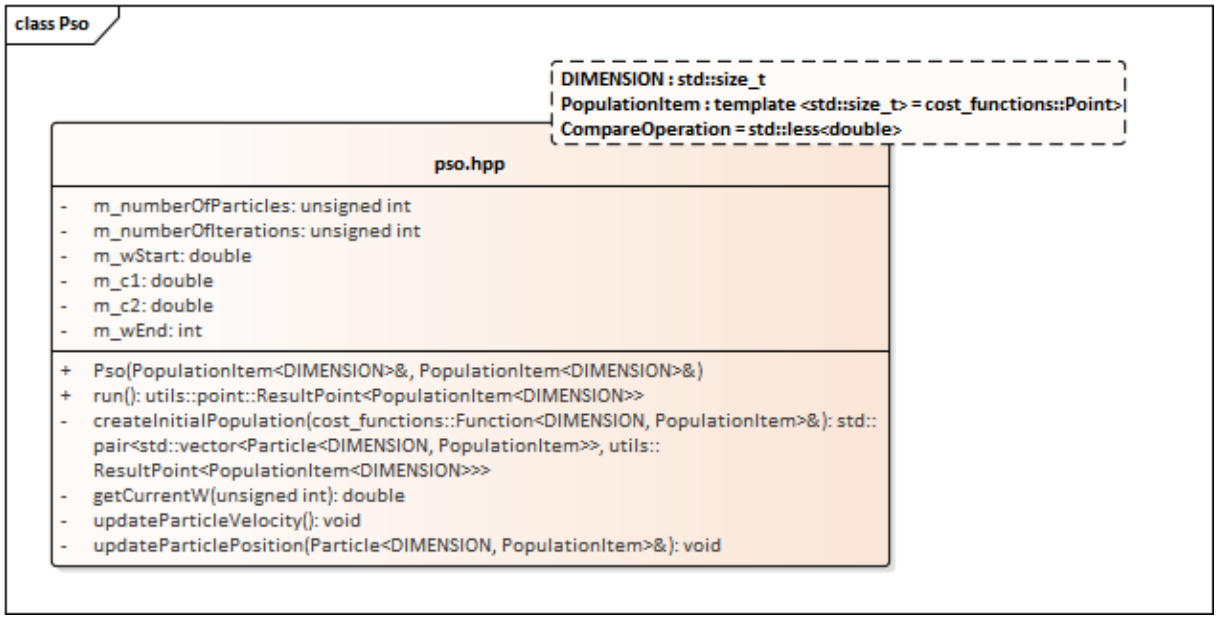
Jednotlivé soukromé metody jsou pak volány postupně ve veřejné metodě run a algoritmus tak prochází všemi typickými fázemi popsanými v předchozích částech tohoto textu. Navíc také dochází k nastavování prostředí, jež je prostředníkem mezi AP a zvoleným metaheuristickým algoritmem. Do prostředí je tedy ukládána např. předchozí pozice jedince a také nejlepší a nejhorší jedinec z populace. Návratová hodnota funkce run je nejlepší nalezené řešení

6.4.4 Implementace algoritmu PSO

Další z metaheuristických algoritmů, jenž byl implementován, je algoritmus PSO ve verzi s lineárně se snižující setrvačností. K implementaci bylo zvoleno řešení s využitím jednotného stylu kódu, a tedy šablony. Toto řešení s sebou nese výhodu, že algoritmus pak dokáže pracovat s různými typy částic. Jak signatury, tak definice metod včetně všech privátních proměnných jsou tedy umístěny v jediném hlavičkovém souboru, a to `pso.hpp`. Metody, jež šablona implementuje jsou uvedeny na Obrázku 13. Soukromé metody jsou pak postupně volány v hlavní veřejné metodě `run()` a algoritmus tak prochází všemi jeho typickými fázemi, jež byly popsány v předchozích částech tohoto textu.

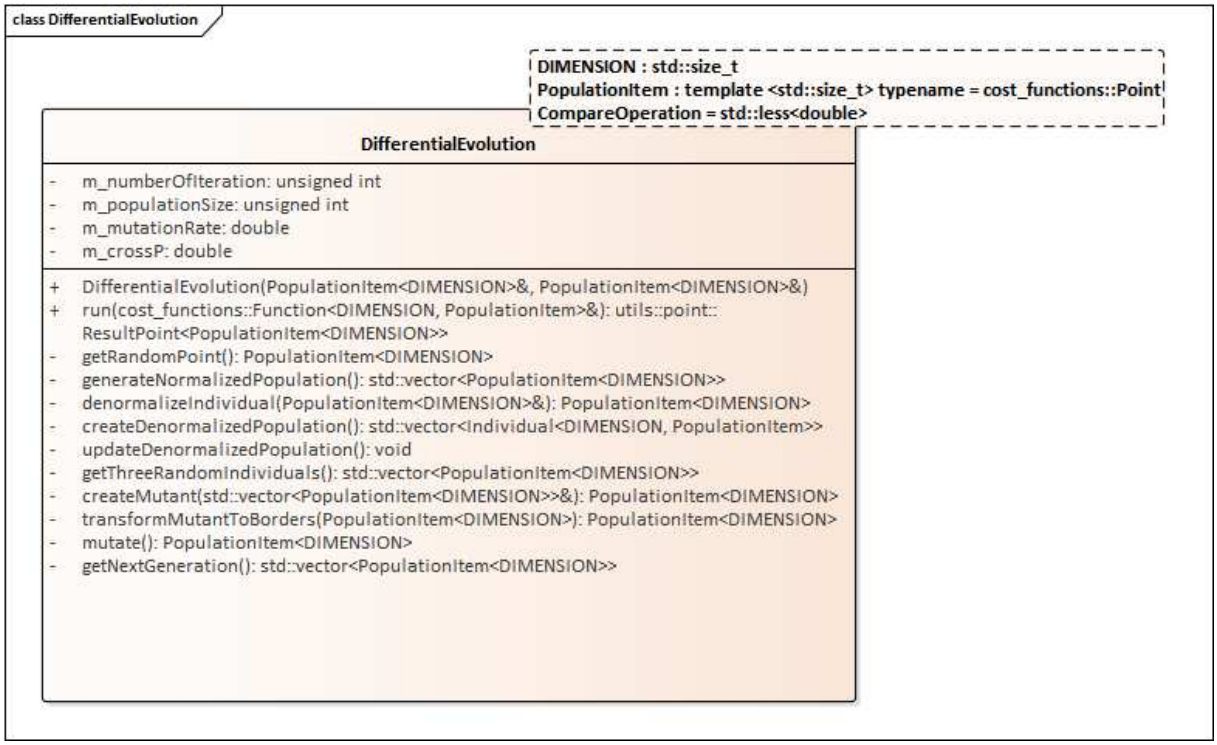
6.4.5 Implementace algoritmu DE

Poslední implementovaný metaheuristický optimalizační algoritmus je evoluční algoritmus diferenciální evoluce. Byla implementována varianta s mutací dle (12) tedy **DE/rand/1**. Aby byl zachován jednotný implementační styl, opět bylo využito výhod šablony. Šablonová třída tak umí pracovat s různými druhy jedinců, a proto je celá implementace opět umístěna v jediném



Obrázek 13: Šablona Pso

hlavičkovém souboru `differentialevolution.hpp`. Struktura šablony je zřejmá z Obrázku 14.



Obrázek 14: Šablona algoritmu diferenciální evoluce

Třída opět implementuje metody pro nastavení např. nejhoršího, nejlepšího jedince v po-

pulaci v dané generaci a také např. předchozí polohu jedince. Tato data jsou pak z prostředí použita k dalšímu výpočtu při vyhodnocování výrazu.

6.5 Analytické programování

Jako metoda symbolické regrese bylo zvoleno Analytické programování. A jelikož analytické programování potřebuje ke své funkci také optimalizační algoritmus, bylo vybráno spojení AP s algoritmem SOMA. Jedinci jsou tedy optimalizováni pomocí algoritmu SOMA a jednotlivé výrazy jsou složeny z jedince pomocí AP.

Pro tvorbu výrazu byly využity funkce s maximálním počtem 2 argumentů a byly uspořádány do 4 množin:

- GFS_0
- GFS_1
- GFS_2
- GFS_{all}

Každá množina GFS je v kódu reprezentována vlastní třídou:

- `Gfs0Member`
- `Gfs1Member`
- `Gfs2Member`

Všechny pak dědí z bázové třídy `GfsNMember`, jež dále dědí z `GfsAllMember`. Důležitou roli hraje také třída `metaheuristics::gfs::MemberFactory`, jež z indexu vytvoří instanci odpovídajícího GFS_n objektu. Tvorbu výrazu z jedince zabezpečuje třída `Expression`, jež třídu `metaheuristics::gfs::MemberFactory` a její metodu `createMember` využívá. Proces probíhá v následujících fázích:

1) Načte se index z jedince.

2) Je vypočtena maximální arita funkce, jež může být vybrána, a tedy vhodné množiny GFS, z níž bude vybíráno dle (17).

$$arity_{max} = distToEnd - debt, \tag{17}$$

kde $arity_{max}$ značí maximální aritu, jež může mít vybraná funkce, $distToEnd$ značí, kolik parametrů v jedinci, z něhož je výraz tvořen, ještě zbývá obsadit, $debt$ značí tzv. dluh neboli, kolik je třeba ještě dosadit parametrů do již zvolených funkcí ve výrazu tak, aby byl výraz validní.

3) Je vybrán člen výrazu z odpovídající GFS množiny a na základě něj je zkonstruován objekt typu `GfsAllMember`, jenž je umístěn do kontejneru typu $std :: vector$ na jeho konec.

4) Proměnná *debt* je navýšena o aritu nově přidaného členu do kontejneru a zároveň snížena o 1 z důvodu nově přidaného členu výrazu. Počet parametrů jedince do konce *distToEnd* je o jedna snížen.

5) Pokud je *debt* vyšší než 0, pak následuje návrat k bodu 1. Pokud podmínka splněna není, pak je tvorba výrazu ukončena.

Z výše popsaného postupu plyne, že jedinec může mít ve srovnání s konečným výrazem vyšší počet parametrů, než má výsledný výraz členů. Vyhodnocení výrazu pak probíhá prostřednictvím třídy `Expression` v metodě `evaluate`, a to v následujících fázích:

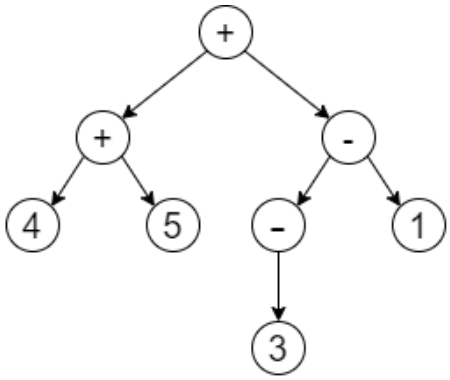
1) Třída metaheuristického algoritmu dostane referenci na výraz složený v AP a zavolá na něm metodu `evaluate`.

2) Výraz ve formě kontejneru `std::vector` je vyhodnocován od konce. Proto je využit zpětný iterátor `std::vector::reverse_iterator`. Načte se tedy první člen od konce, dojde k jeho přetypování na správnou třídu GFS dle jeho arity a zavolá se na něm opět metoda `evaluate`. Každá GFS třída má metodu `evaluate` implementovanou odpovídajícím způsobem a přijímá dle své arity příslušný počet parametrů.

3) Výsledek je uložen do pomocného kontejnerů mezivýsledků typu `std::vector`. Tyto mezivýsledky jsou dále použity jako vstup pro metodu `evaluate` u členů, pro které platí ($arita > 0$). Pokud je některý člen z kontejneru mezivýsledků poskytnut do metody `evaluate`, pak je ukazatel posunut na další mezivýsledek bezprostředně za použitým členem a nový výsledek metody `evaluate` je vložen na konec kontejneru.

4) Po vyhodnocení posledního členu výrazu je odečten celkový výsledek, který odpovídá poslednímu členu v kontejneru mezivýsledků.

Pro lepší pochopení vyhodnocovací strategie je přiložen následující příklad vyhodnocení konkrétního výrazu. Mějme výraz jako na Obrázku 15. Pro lepší představu je zde zobrazen výraz také formou stromu a pod ním je interpretace výrazu p ve formě kontejneru.



p(plus, plus, minus, 4, 5, unární minus, 1, 3)

Obrázek 15: Výraz ve formě stromu

Jak již bylo řečeno, výraz je vyhodnocován pozpátku, je tedy vyčten první prvek od konce - číslo 3. Jeho vyhodnocením je tedy hodnota 3, která je umístěna do pomocného kontejneru $temp(3)$. Jako další je druhý prvek od konce výrazu - číslo 1. Jeho vyhodnocením je hodnota 1 a je přidána na konec pomocného kontejneru mezivýsledků $temp(3, 1)$. Třetí prvek od konce výrazu obsahuje operaci unární minus, jež má aritu jedna a potřebuje tedy pro své vyhodnocení jeden argument. Jako tento argument je použit první člen v kontejneru mezivýsledků, čímž je číslo 3. Dostáváme tak hodnotu -3, jež je opět uložena do kontejneru mezivýsledků $temp(3, 1, -3)$. Další prvek za unárním minus je číslo 5. Do kontejneru pro mezivýsledky je tedy uložena hodnota 5: $temp(3, 1, -3, 5)$. Dalším prvkem je číslo 4, do kontejneru pro mezivýsledky tedy uložíme hodnotu 4: $temp(3, 1, -3, 5, 4)$. Jako další následuje ve výraze binární operace minus, jež pro své vyhodnocení potřebuje dva parametry. Z pole mezivýsledků již byla vyčerpána hodnota první hodnota proto jako parametry poskytneme následující dvě hodnoty, tedy 1 a -3. Výsledkem operace je -4, což je opět zapsáno do kontejneru mezivýsledků $temp(3, 1, -3, 5, 4, -4)$. Další operací, jež ve výrazu následuje je binární plus. Tato funkce opět přijímá dva parametry, a proto jsou vyčteny další dvě hodnoty z kontejneru mezivýsledků 5 a 4. Je vrácena hodnota 9, jež je opět zapsána na konec kontejneru mezivýsledků $temp(3, 1, -3, 5, 4, -4, 9)$. Další operací ve výrazu je operace plus, jež má dva parametry, jsou tedy vyčteny další dvě hodnoty z pole mezivýsledků -4 a 9. Jako výsledek je vrácena hodnota 5, která je opět vložena na konec kontejneru mezivýsledků $temp(3, 1, -3, 5, 4, -4, 9, 5)$. Další funkce již ve výrazu nejsou, proto je výpočet ukončen a je vrácena poslední hodnota v kontejneru mezivýsledků. Tou je číslo 5. Výsledek celého výrazu je tedy číslo 5.

6.6 Nastavení

V této části se nacházejí nastavení programu, a to jak kompilační konstanty, tak nastavení, která může uživatel ovlivnit při spuštění. Mezi kompilační konstanty se řadí například dimenze účelových funkcí, dimenze jedinců v AP. Konstanty jsou součástí struktury `CompilationSettings`. Mezi nastavení, jež může ovlivnit uživatel při spuštění programu, patří typ metaheuristického algoritmu, jenž se má spustit a účelová funkce, jejíž extrém se má hledat. Do této sekce nastavení spadá také výraz, s nímž má být metaheuristický algoritmus spuštěn.

V této sekci se nachází také komponenta pro přístup k nastavením zmíněných výše, jež je navržena podle návrhového vzoru jedináček. Zároveň tato komponenta obsahuje logiku pro zpracování parametrů spuštění zkompilovaného programu.

6.7 Pomocné knihovny

Tato část zdrojového kódu slouží jako kolekce funkcí a tříd, jež jsou využívány v různých částech programu. Mezi nejdůležitější komponenty, jež se zde nachází, patří třída reprezentující N-dimenzionální bod prohledávaného prostoru a generátor náhodných čísel. Tato část dále ob-

sahuje pomocné funkce usnadňující práci s typovým systémem jazyka C++, s výčtovými typy nebo s poli.

Bod je reprezentován třídou `Point`, jejíž implementace poskytuje podobné rozhraní jako kontejnery ze standardní šablonové knihovny. Dále obsahuje přetížení operátorů a volatelných objektů ze standardní knihovny, které s body umožňují provádět základní matematické operace jako například sčítání a odčítání.

Generátor náhodných čísel je komponenta, jež implementuje rozhraní pro získání náhodně inicializovaných objektů různého typu. Mezi metody, jež tato třída poskytuje, patří například metoda pro získání náhodného souseda, jež podléhá normálnímu rozdělení, metoda pro získání náhodného celého čísla, metoda pro získání čísla s plovoucí desetinnou čárkou nebo metoda pro získání náhodně inicializovaného objektu typu `Point` z dané oblasti. Jako generátor náhodných čísel je použita implementace MT19937 ze standardní knihovny.

7 Experimenty

Cílem této práce je naleznout vhodný výraz, jenž by zlepšil optimalizační schopnosti metaheuristických algoritmů zmíněných v předchozích částech této práce. Jaké experimenty byly provedeny, bude popsáno v následující části.

7.1 Analytické programování

Pro sestavování výrazu v analytickém programování byly využity jak funkce, tak konstanty. Jejich přehled je uveden v Tabulce 6. Jedinci složení z indexů do množiny GFS_{all} byli následně

Tabulka 6: Operace a konstanty použité v AP

Funkce	Arita	Popis
$plus$	2	operace sčítání
$minus$	2	operace odčítání
$násobení$	2	operace násobení
$minus_{unární}$	1	operace unární minus
$sinus$	1	
$cosinus$	1	
$\vec{x}^{g}_{best}$	0	nejlepší prvek v dané generaci
$\vec{x}^{g}_{worst}$	0	nejhorší prvek v dané generaci
$\vec{x}^{g}_{R_1}$	0	náhodný prvek 1 v dané generaci
$\vec{x}^{g-1}_{R_1}$	0	stav náhodného prvku 1 v předešlé generaci
$\vec{x}^{g}_{R_2}$	0	náhodný prvek 2 v dané generaci
$\vec{x}^{g-1}_{R_2}$	0	stav náhodného prvku 2 v předešlé generaci
$\vec{x}^{g-1}$	0	stav prvku v předešlé generaci
K	0	náhodné číslo, pro každého jedince a každou iteraci jiné

optimalizováni pomocí SOMA algoritmu. Nastavení parametrů pro běh SOMA algoritmu ve spojení s AP je uveden v Tabulce 7.

Tabulka 7: Hodnoty parametrů SOMA s AP

Parametr	Hodnota	Popis
D	10	dimenze
$PathLength$	3,2	délka cesty
$Step$	0,41	krok
PRT	0,3	parametr perturbace
$PopSize$	15	velikost populace
$migrations$	40	počet migračních kol

Jako účelová funkce pro určení vhodnosti jednotlivých výrazů byly voleny tři metaheuristické algoritmy - PSO, SOMA a DE. Aby mělo ohodnocení jedince vyšší vypovídající hodnotu, byl

metaheuristický algoritmus pro daný výraz spuštěn více-krát po sobě a z výsledných nalezených optim byl vypočítám aritmetický průměr. Tyto metaheuristické algoritmy byly spouštěny pro 5 vybraných účelových funkcí s přihlédnutím k jejich průběhu:

- Sférická funkce
- Schwefelova funkce
- Ackleyho funkce
- Levyho funkce
- Michalewiczova funkce

Parametry PSO algoritmu byly nastaveny na hodnoty dle Tabulky 8. Algoritmus byl pro

Tabulka 8: Hodnoty parametrů PSO

Parametr	Hodnota	Popis
particles	40	počet část v hejnu
c_1	2	učící faktor
c_2	2	učící faktor
iterace	75	počet iterací algoritmu
w_{start}	0,9	počáteční hodnota setrvačnosti
w_{end}	0,4	koncová hodnota setrvačnosti

každé ohodnocení výrazu AP spuštěn třikrát a výsledek následně průměrován pomocí *aritmetického průměru*. Výrazem vytvořeným v AP byl modifikován výpočet rychlosti částic dle (18).

$$\vec{v}_i^{it+1} = w \cdot \vec{v}_i^{it} + c_1 \cdot rand_1 \cdot (\vec{pBest}_i^{it} - \vec{x}_i^{it}) + c_2 \cdot rand_2 \cdot (\vec{gBest} - \vec{x}_i^{it}) + výraz, \tag{18}$$

kde it prezentuje migrační kolo, i značí pořadí částice v hejnu, $\vec{x}_i^{it}$ představuje pozici částice v daném kroku, $\vec{v}_i^{it}$ značí rychlost částice v daném kroku, $\vec{v}_i^{it+1}$ je rychlost částice v následujícím kroku. Nejlepší poloha částice s indexem i je pak uložena v proměnné $\vec{pBest}_i^{it}$ a nejlepší poloha celého hejna je reprezentována hodnotou $\vec{gBest}$. $rand_1$ a $rand_2$ jsou pak náhodně generované hodnoty v rozmezí [0, 1], c_1 a c_2 jsou učící faktory a w je setrvačnost. Výraz vytvořený v AP je pak přičten na konci a zde symbolicky reprezentován jako $výraz$.

Parametry SOMA metaheuristického algoritmu byly nastaveny dle Tabulky 9. Algoritmus byl spuštěn pro každé ohodnocení výrazu sestaveného v AP vždy pětkrát, následně byl vytvořen *aritmetický průměr* a použit jako vhodnost. Výraz byl v algoritmu SOMA použit při výpočtu nové polohy jedince. Využití výrazu je znázorněno v (19).

$$\vec{x}_i^{ML+1} = \vec{x}_{i,start}^{ML} + (\vec{x}_L^{ML} - \vec{x}_{i,start}^{ML} + výraz) \cdot t \cdot PRTVector_i^{ML}, \tag{19}$$

Tabulka 9: Hodnoty parametrů SOMA

Parametr	Hodnota	Popis
D	10	dimenze
$PathLength$	3,0	délka cesty
$Step$	0,33	krok
PRT	0,1	parametr perturbace
$PopSize$	15	velikost populace
$migrations$	50	počet migračních kol

kde i označuje pořadí jedince v populaci, ML pořadí migračního kola, $\vec{x}_i^{ML+1}$ novou polohu jedince, $\vec{x}_{i,start}^{ML}$ startovní polohu jedince, $\vec{x}_L^{ML}$ polohu lídra, $PathLength]$ a $PRTVector_i^{ML}$ značí perturbační vektor pro daného jedince a dané migrační kolo, výraz sestavený pomocí AP je v rovnici reprezentován jako *výraz* a pro t platí, že $t \in [0, \text{po } Step \text{ az po}$.

Diferenciální evoluce, jež byla použita jako třetí metaheuristický algoritmus pro ohodnocení vhodnosti výrazů sestaveného v AP, byla spouštěna s parametry, jež jsou uvedeny v Tabulce 10. Pro každé ohodnocení jedince je DE algoritmus spuštěn desetkrát a jako ohodnocení je pou-

Tabulka 10: Hodnoty parametrů DE

Parametr	Hodnota	Popis
D	10	dimenze
CR	0,9	práh křížení
F	0,5	mutační konstanta
$PopSize$	50	velikost populace
$generations$	60	počet generací

žit *aritmetický průměr* všech dosažených výsledků. V případě algoritmu DE byl výraz generovaný pomocí AP využit ve fázi mutace pro vytváření šumového vektoru. Použití výrazu je viditelné v (20).

$$\vec{v}_i^g = výraz, \tag{20}$$

kde $\vec{v}_i^g$ značí výsledný šumový vektor, výraz z AP je zde reprezentován jako $výraz$.

7.2 Vyhodnocení získaných výsledků

Analytické programování s metaheuristickým algoritmem vyhledávájícím optimální výraz byl pro každou účelovou funkci a pro každý metaheuristický algoritmus spuštěn více-krát a byl vybrán výraz s nejlepším ohodnocením. Počet spuštění bylo pro každý algoritmus jiné s ohledem na délku jeho běhu, pro algoritmus SOMA byl počet běhů 5, pro algoritmus PSO byl počet běhů 3 a pro algoritmus DE běželo AP 10-krát. Výsledné nalezené nejlépe ohodnocené výrazy jsou uvedeny v Tabulce 11.

Tabulka 11: Výsledné nalezené nejlépe ohodnocené výrazy

Algoritmus	Výraz
SOMA	$-(\vec{x}^{g-1} + \vec{x}^{g-1}_{R_1})$
PSO	$\vec{x}^{g-1}$
DE	$(\vec{x}^{g}_{R_1} - \vec{x}^{g-1}_{R_1}) \cdot 1,0 + \vec{x}^{g}_{R_2}$

Následně pro srovnání byl každý z metaheuristických algoritmů spuštěn bez nalezeného výrazu a s nalezeným výrazem, a to vždy 31-krát pro každou testovací funkci uvedenou v Tabulce 5. Výsledky byly vyhodnoceny statisticky pomocí Wilcoxon rank-sum testu s parametrem $\alpha = 0,05$, kde v případě zlepšení je uveden znak $+$, v případě zhoršení znak $-$ a v případě, kdy nedojde k žádnému posunu ve výsledcích, je uveden symbol $=$. Vyhodnocení výsledků pro jednotlivé metaheuristické algoritmy je pak zpracováno v oddělených tabulkách. Výsledky běhu algoritmu SOMA jsou zpracovány v Tabulce 12. Zde došlo ke zlepšení téměř na všech testovacích funkcí kromě Zakharovovy funkce, kde ovšem nedošlo ani ke zhoršení a výsledky optimalizace jsou srovnatelné jak pro běh SOMA s nově nalezeným výrazem, tak pro běh standardní verze bez výrazu.

Vyhodnocení výsledků pro algoritmus PSO je zpracováno v Tabulce 13. Na výsledcích je patrné, že v případě tohoto metaheuristického algoritmu se nepodařilo dosáhnout takového úspěchu jako u algoritmu SOMA. Ke zlepšení optimalizačních schopností došlo v případě Ackleyho testovací funkce. Nezměněné optimalizační schopnosti měl algoristmus PSO s použitím nalezeného výrazu pro Griewankovu, Levyho, Rastriginovu, Rosenbrockovu, Sférickou a Zakharovu testovací funkci. Naopak k mírnému zhoršení výsledků došlo v případě dvou testovacích funkcí, a to Michalewiczovu a Schwefelovu.

Výsledky běhů algoritmu DE s výrazem a bez výrazu jsou shrnuty v Tabulce 14. Ani zde se nepodařilo dosáhnout takového úspěchu jako v případě metaheuristického algoritmu SOMA, nicméně ke zlepšení optimalizace algoritmu DE došlo v případě dvou testovacích funkcí, a to v případě funkce Ackleyho a Schwefelovy. Na ostatních testovacích funkcích došlo ke zhoršení.

Tabulka 12: Vyhodnocení běhu algoritmu SOMA s výrazem a bez něj

	SOMA s výrazem						SOMA bez výrazu				
	min	max	mean	median	std		min	max	mean	median	std
Ackley	5,88E+00	1,26E+01	8,47E+00	8,46E+00	1,70E+00	+	1,81E+01	2,01E+01	1,95E+01	1,97E+01	4,38E-01
Griewank	3,13E-01	6,34E-01	4,68E-01	4,61E-01	8,34E-02	+	1,13E+00	1,52E+00	1,34E+00	1,34E+00	9,76E-02
Levy	2,02E+00	1,83E+01	8,73E+00	8,79E+00	3,55E+00	+	1,14E+02	3,76E+02	2,90E+02	2,87E+02	6,39E+01
Michalewicz	-6,43E+00	-5,22E+00	-5,89E+00	-5,90E+00	2,71E-01	+	-5,34E+00	-3,80E+00	-4,71E+00	-4,73E+00	3,52E-01
Rastrigin	2,90E+01	7,39E+01	5,21E+01	5,10E+01	9,58E+00	+	1,09E+03	2,37E+03	1,66E+03	1,71E+03	3,43E+02
Rosenbrock	1,29E+03	1,66E+04	7,07E+03	5,87E+03	4,21E+03	+	9,60E+06	7,82E+07	3,97E+07	4,14E+07	1,70E+07
Schwefel	3,92E+03	3,96E+03	3,94E+03	3,94E+03	7,37E+00	+	3,94E+03	3,99E+03	3,97E+03	3,97E+03	1,41E+01
Sphere	1,43E+00	1,94E+01	6,36E+00	5,94E+00	3,79E+00	+	6,40E+02	2,35E+03	1,55E+03	1,64E+03	4,43E+02
Zakharov	1,31E+03	3,86E+03	3,06E+03	3,20E+03	6,70E+02	=	1,53E+03	4,33E+03	3,05E+03	3,22E+03	7,84E+02

Tabulka 13: Vyhodnocení běhu algoritmu PSO s výrazem a bez něj

	PSO s výrazem						PSO bez výrazu				
	min	max	mean	median	std		min	max	mean	median	std
Ackley	1,03E-01	2,00E+01	1,94E+01	2,00E+01	3,57E+00	+	3,08E+00	2,00E+01	1,95E+01	2,00E+01	3,04E+00
Griewank	5,10E-07	1,26E+00	2,67E-01	5,75E-02	3,51E-01	=	4,94E-02	3,79E-01	1,84E-01	1,99E-01	7,35E-02
Levy	3,08E+00	1,04E+03	1,51E+02	8,72E+00	2,45E+02	=	3,05E-01	1,64E+02	2,11E+01	1,18E+01	3,13E+01
Michalewicz	-3,13E+00	-1,76E+00	-2,29E+00	-2,26E+00	3,53E-01	-	-4,14E+00	-2,57E+00	-3,22E+00	-3,15E+00	3,64E-01
Rastrigin	3,83E-03	2,52E+03	5,97E+02	4,29E+01	9,57E+02	=	3,18E+01	1,09E+02	6,23E+01	5,38E+01	2,29E+01
Rosenbrock	8,97E+00	2,50E+05	1,08E+05	9,45E+03	1,23E+05	=	1,89E+01	2,50E+05	4,10E+04	1,31E+03	8,39E+04
Schwefel	3,84E+03	4,04E+03	3,90E+03	3,91E+03	5,12E+01	-	3,84E+03	3,86E+03	3,84E+03	3,84E+03	5,42E+00
Sphere	5,51E-08	2,50E+03	7,34E+02	2,00E-01	1,12E+03	=	9,02E-02	5,55E+00	1,09E+00	3,95E-01	1,29E+00
Zakharov	2,99E+03	1,52E+04	8,42E+03	7,78E+03	3,19E+03	=	1,89E+01	3,41E+03	7,85E+02	2,92E+02	1,11E+03

Tabulka 14: Vyhodnocení běhu algoritmu DE s výrazem a bez něj

	DE s výrazem						DE bez výrazu				
	min	max	mean	median	std		min	max	mean	median	std
Ackley	1,42E+00	2,21E+00	1,86E+00	1,89E+00	1,94E-01	+	1,37E+01	2,00E+01	1,84E+01	1,93E+01	1,83E+00
Griewank	1,42E+00	2,21E+00	1,86E+00	1,89E+00	1,94E-01	-	3,90E-01	8,14E-01	6,41E-01	6,60E-01	9,71E-02
Levy	4,45E+02	1,57E+03	8,94E+02	8,51E+02	2,72E+02	-	1,49E+00	1,04E+01	3,59E+00	2,75E+00	2,20E+00
Michalewicz	-3,08E+00	-1,49E+00	-2,14E+00	-2,11E+00	4,40E-01	-	-4,83E+00	-3,28E+00	-3,89E+00	-3,75E+00	4,27E-01
Rastrigin	1,78E+03	4,91E+03	3,58E+03	3,67E+03	7,73E+02	-	5,09E+01	8,08E+01	6,50E+01	6,65E+01	8,09E+00
Rosenbrock	3,52E+07	3,90E+08	2,33E+08	2,44E+08	8,85E+07	-	1,30E+02	3,31E+03	9,70E+02	8,54E+02	6,53E+02
Schwefel	3,84E+03	3,84E+03	3,84E+03	3,84E+03	0,00E+00	+	3,84E+03	3,86E+03	3,84E+03	3,84E+03	8,03E+00
Sphere	1,41E+03	4,77E+03	3,35E+03	3,38E+03	8,32E+02	-	7,03E-01	5,70E+00	2,26E+00	2,07E+00	1,22E+00
Zakharov	4,37E+03	2,57E+04	1,14E+04	9,47E+03	6,27E+03	-	6,77E+01	4,81E+02	2,59E+02	2,74E+02	1,08E+02

8 Závěr

Cílem této práce je zlepšení optimalizačních schopností tří metaheuristických algoritmů. Výsledky, jež byly prezentovány v předchozích částech této práce, ukazují, že určitá zlepšení se podařilo nalézt. U některých algoritmů jsou tato zlepšení výraznější a více univerzální, například v případě algoritmu SOMA. U některých algoritmů došlo ke zlepšení jen pro určité konkrétní problémy.

Výrazného zlepšení se podařilo nalézt pro algoritmus SOMA. Dle výsledků, jež jsou znázorněny v Tabulce 12, je viditelné, že optimalizační schopnosti algoritmu dosáhly lepších řešení pro všechny testovací funkce kromě jedné. U této - Zakharovy - funkce došlo ke stejným výsledkům jako bez využití výrazu nalezeného prostřednictvím SOMA s AP.

V případě PSO algoritmu výsledky nedosahovaly takových úspěchů jako u SOMA, avšak i zde se podařilo nalézt jistého zlepšení. Bylo tomu tak u testovací Ackleyho funkce. K mírnému zhoršení optimalizačních schopností PSO došlo u dvou testovacích funkcí, a to funkce Schwefelovy a Michalewiczovy. V případě běhu PSO s využitím ostatních testovacích funkcí nebylo pozorováno žádných posunů ve výsledcích, tedy ani k lepšímu, ani k horšímu. Výsledky jsou zpracovány v Tabulce 13.

Posledním metaheuristickým algoritmem, jenž byl součástí experimentů a bylo usilováno o vylepšení jeho optimalizačních schopností, byl algoritmus DE. Také zde se podařilo jistých úspěchů dosáhnout, ačkoliv ne tak příznivých jako u algoritmu SOMA. Konkrétně zlepšení nastalo pro dvě testovací funkce - Ackleyho a Schwefelovu. U zbytku testovacích účelových funkcí bylo pozorováno zhoršení optimalizačních schopností DE. Výsledky běhů s využitím výrazu a bez něj jsou zpracovány v samostatné Tabulce 14.

Jev, který je na výsledcích pozorovatelný, lze vysvětlit prostřednictvím teorie, s níž se v literatuře setkáváme pod názvem „No Free Lunch Teorém“, mnohdy ve formě zkratky jako NFL [1]. Jak je znatelné, u PSO a DE byly výsledky znatelně horší než u SOMA. Toto mohlo být také ovlivněno tím, že výsledný výraz byl vyhledáván na pěti účelových funkcích zmíněných v předešlých částech textu a nejlepší zástupce byl pak vybrán pouze jeden. Ten byl následně aplikován pro všechny účelové funkce. Tohoto řešení bylo využito s přihlédnutím na výpočetní a časové možnosti této práce, neboť i přes malé pokrytí testovacích funkcí a zjevně nízký počet spuštění (5 běhů AP pro každou účelovou funkci), trval celý experiment 144 hodin. V případě lepšího technického vybavení o větším počtu výpočetních jader či s vyšším počtem výpočetních strojů by mohl být experiment rozšířen o více účelových funkcí a více výrazů, čímž by došlo k nalezení tzv. „řešení na míru“, jenž by pravděpodobně bylo úspěšnější. Algoritmy by také bylo možné spouštět s vyšším počtem generací či jedinců a byla by tak vyšší pravděpodobnost nalezení optimálnějšího řešení. Celý problém by se také dal řešit s využitím více-účelové optimalizace, jež by mohla nalezené výsledky dále zlepšit. Jistá zlepšení také nabízí využití paralelizace a výpočetních technologií, jež ji podporují, tímto je například technologie CUDA.

I přesto, že experiment tedy běžel v omezených podmínkách a skýtá jisté možnosti pro budoucí vylepšení, bylo v závěru dosaženo příznivých výsledků, jež mají na poli optimalizace nemalý význam.

Literatura

1. ZELINKA, Ivan; OPLATKOVÁ, Zuzana; OŠMERA, Pavel; ŠEDA, Miloš; VČELAŘ, František. *Evoluční výpočetní techniky: principy a aplikace*. BEN, 2008.
2. DIEP, Quoc Bao; ZELINKA, Ivan; DAS, Swagatam. Self-organizing migrating algorithm pareto. In: *MENDEL*. 2019, sv. 25, s. 111–120. Č. 1.
3. TRUONG, Thanh Cong; DIEP, Quoc Bao; ZELINKA, Ivan; SENKERIK, Roman. Pareto-Based Self-organizing Migrating Algorithm Solving 100-Digit Challenge. In: *Swarm, Evolutionary, and Memetic Computing and Fuzzy and Neural Computing*. Springer, 2019, s. 13–20.
4. SANTOS COELHO, Leandro dos; ALOTTO, Piergiorgio. Electromagnetic optimization using a cultural self-organizing migrating algorithm approach based on normative knowledge. *IEEE Transactions on Magnetics*. 2009, roč. 45, č. 3, s. 1446–1449.
5. SANTOS COELHO, Leandro dos; MARIANI, Viviana Cocco. An efficient cultural self-organizing migrating strategy for economic dispatch optimization with valve-point effect. *Energy Conversion and Management*. 2010, roč. 51, č. 12, s. 2580–2587.
6. REYNOLDS, RG; CHUNG, C. The use of cultural algorithms to evolve multiagent cooperation. In: *Proc. Micro-Robot World Cup Soccer Tournament*. 1996, s. 53–56.
7. VOLNÁ, Eva. Evoluční algoritmy a neuronové sítě. *Ostrava: Ostravská univerzita v Ostravě*. 2013.
8. ZELINKA, Ivan. SOMA—self-organizing migrating algorithm. In: *Self-Organizing Migrating Algorithm*. Springer, 2016, s. 3–49.
9. DEVENDRA, D; ZELINKA, I. Self-organizing migrating algorithm: methodology and implementation. In: *Springer-Verlag Berlin Heidelberg*. 2016.
10. LAURET, Philippe; BOYER, Harry; RIVIERE, Carine; BASTIDE, Alain. A genetic algorithm applied to the validation of building thermal models. *Energy and buildings*. 2005, roč. 37, č. 8, s. 858–866.
11. DIEP, Quoc Bao; ZELINKA, Ivan; DAS, Swagatam. Self-organizing migrating algorithm for the 100-digit challenge. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2019, s. 3–4.
12. TOMASZEK, Lukas; ZELINKA, Ivan; CHADLI, Mohammed. On the Leader Selection in the Self-Organizing Migrating Algorithm. In: *MENDEL*. 2019, sv. 25, s. 171–178. Č. 1.
13. DEEP, Kusum et al. A new hybrid self organizing migrating genetic algorithm for function optimization. In: *2007 IEEE Congress on Evolutionary Computation*. 2007, s. 2796–2803.
14. DEEP, Kusum et al. A self-organizing migrating genetic algorithm for constrained optimization. *Applied Mathematics and Computation*. 2008, roč. 198, č. 1, s. 237–250.

15. DAVIS, Lawrence. Handbook of genetic algorithms. 1991.

16. KADLEC, Petr; RAIDA, Zbyněk. A Novel Multi-Objective Self-Organizing Migrating Algorithm. *Radioengineering*. 2011, roč. 20, č. 4.

17. KADLEC, Petr; RAIDA, Zbyněk. Comparison of novel multi-objective self organizing migrating algorithm with conventional methods. In: *Proceedings of 21st International Conference Radioelektronika 2011*. 2011, s. 1–4.

18. KENNEDY, James; EBERHART, Russell. Particle swarm optimization. In: *Proceedings of ICNN'95-International Conference on Neural Networks*. 1995, sv. 4, s. 1942–1948.

19. SHI, Yuhui; EBERHART, Russell C. Parameter selection in particle swarm optimization. In: *International conference on evolutionary programming*. 1998, s. 591–600.

20. CLERC, Maurice. The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. In: *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*. 1999, sv. 3, s. 1951–1957.

21. IMRAN[a], Muhammad; HASHIMA, Rathiah; KHALIDB, Noor Elaiza Abd. An overview of particle swarm optimization variants. *Procedia Engineering*. 2013, roč. 53, s. 491–496.

22. EBERHART, Russ C; SHI, Yuhui. Comparing inertia weights and constriction factors in particle swarm optimization. In: *Proceedings of the 2000 congress on evolutionary computation. CEC00 (Cat. No. 00TH8512)*. 2000, sv. 1, s. 84–88.

23. CHONGPENG, Huang; YULING, Zhang; DINGGUO, Jiang; BAOGUO, Xu. On some non-linear decreasing inertia weight strategies in particle swarm optimization. In: *2007 Chinese Control Conference*. 2007, s. 750–753.

24. SHI, Yuhui; EBERHART, Russell C. Fuzzy adaptive particle swarm optimization. In: *Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546)*. 2001, sv. 1, s. 101–106.

25. WANG, Dongshu; TAN, Dapei; LIU, Lei. Particle swarm optimization algorithm: an overview. *Soft Computing*. 2018, roč. 22, č. 2, s. 387–408.

26. ANGELINE, Peter J. Evolutionary optimization versus particle swarm optimization: Philosophy and performance differences. In: *International Conference on Evolutionary Programming*. 1998, s. 601–610.

27. LOVBJERG, Morten; RASMUSSEN, Thomas Kiel; KRINK, Thiemo et al. Hybrid particle swarm optimiser with breeding and subpopulations. In: *Proceedings of the genetic and evolutionary computation conference*. 2001, sv. 24, s. 469–476.

28. ANGELINE, Peter J. Using selection to improve particle swarm optimization. In: *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98TH8360)*. 1998, s. 84–89.

29. CHEN, Ying; FENG, Yong; LI, Xinyang. A parallel system for adaptive optics based on parallel mutation PSO algorithm. *Optik*. 2014, roč. 125, č. 1, s. 329–332.

30. TSAFARAKIS, Stelios; SARIDAKIS, Charalampos; BALTAS, George; MATSATSINIS, Nikolaos. Hybrid particle swarm optimization with mutation for optimizing industrial product lines: An application to a mixed solution space considering both discrete and continuous design variables. *Industrial Marketing Management*. 2013, roč. 42, č. 4, s. 496–506.

31. MENG, Anbo; LI, Zhuan; YIN, Hao; CHEN, Sizhe; GUO, Zhuangzhi. Accelerating particle swarm optimization using crisscross search. *Information Sciences*. 2016, roč. 329, s. 52–72.

32. VLACHOGIANNIS, John G; LEE, Kwang Y. Economic load dispatch—A comparative study on heuristic optimization techniques with an improved coordinated aggregation-based PSO. *IEEE Transactions on Power Systems*. 2009, roč. 24, č. 2, s. 991–1001.

33. SELVAKUMAR, A Immanuel; THANUSHKODI, K. Optimization using civilized swarm: solution to economic dispatch with multiple minima. *Electric Power Systems Research*. 2009, roč. 79, č. 1, s. 8–16.

34. SHIN, Young-Bin; KITA, Eisuke. Search performance improvement of particle swarm optimization by second best particle information. *Applied Mathematics and Computation*. 2014, roč. 246, s. 346–354.

35. STORN, Rainer; PRICE, Kenneth et al. Differential evolution–a simple and efficient adaptive scheme for global optimization over continuous spaces: technical report TR-95-012. *International Computer Science, Berkeley, California*. 1995.

36. YANG, SH; NATARAJAN, U; SEKAR, M; PALANI, S. Prediction of surface roughness in turning operations by computer vision using neural network trained by differential evolution algorithm. *The International Journal of Advanced Manufacturing Technology*. 2010, roč. 51, č. 9-12, s. 965–971.

37. DAS, Swagatam; MULLICK, Sankha Subhra; SUGANTHAN, Ponnuthurai N. Recent advances in differential evolution–an updated survey. *Swarm and Evolutionary Computation*. 2016, roč. 27, s. 1–30.

38. NOMAN, Nasimul; BOLLEGALA, Danushka; IBA, Hitoshi. An adaptive differential evolution algorithm. In: *2011 IEEE Congress of Evolutionary Computation (CEC)*. 2011, s. 2229–2236.

39. GEORGIOUDAKIS, Manolis; PLEVRIS, Vagelis. On the Performance of Differential Evolution Variants in Constrained Structural Optimization. *Procedia Manufacturing*. 2020, roč. 44, s. 371–378.

40. TANABE, Ryoji; FUKUNAGA, Alex. Success-history based parameter adaptation for differential evolution. In: *2013 IEEE congress on evolutionary computation*. 2013, s. 71–78.

41. BREST, Janez; GREINER, Sao; BOSKOVIC, Borko; MERNIK, Marjan; ZUMER, Viljem. Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *IEEE transactions on evolutionary computation*. 2006, roč. 10, č. 6, s. 646–657.

42. ZHANG, Jingqiao; SANDERSON, Arthur C. JADE: Self-adaptive differential evolution with fast and reliable convergence performance. In: *2007 IEEE congress on evolutionary computation*. 2007, s. 2251–2258.

43. POIKOLAINEN, Ilpo; NERI, Ferrante; CARAFFINI, Fabio. Cluster-based population initialization for differential evolution frameworks. *Information Sciences*. 2015, roč. 297, s. 216–235.

44. ZELINKA, Ivan; OPLATKOVA, Zuzana; NOLLE, Lars. Analytic programming–Symbolic regression by means of arbitrary evolutionary algorithms. *International Journal of Simulation: Systems, Science and Technology*. 2005, roč. 6, č. 9, s. 44–56.

45. BANZHAF, Wolfgang; NORDIN, Peter; KELLER, Robert E; FRANCONE, Frank D. *Genetic programming*. Springer, 1998.

46. RYAN, Conor; COLLINS, John James; NEILL, Michael O. Grammatical evolution: Evolving programs for an arbitrary language. In: *European Conference on Genetic Programming*. 1998, s. 83–96.

A Seznam přiložených souborů

- 2020_CER585_DP.pdf - Textová část diplomové práce
- 2020_CER585_DP.zip - Zdrojový kód programu diplomové práce